Valter Balegas de Sousa

Mestre em Engenharia Informática

Invariant preservation in geo-replicated data stores

Dissertação para obtenção do Grau de Doutor em

Informática

Orientador: Nuno Preguiça, Associate Professor,
NOVA LINCS, FCT, Universidade NOVA de Lisboa

Júri

Presidente:	Luís Caires
Arguentes:	Willy Zwaenepoel
	Luís Manuel A. Veiga
Vogais:	José Orlando Pereira
	Nuno Manuel Ribeiro Preguiça
	João Carlos A. Leitão

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2017

Invariant preservation in geo-replicated data stores

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "unlthesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

Acknowledgements

This thesis was only possible with with the hard work of all the people that were involved in it. From this work, and time spent with it, I will take knowledge and personal experiences for all my professional and adult life.

First, and foremost, I want to thank my advisor, Nuno Preguiça, in the development of this work. Nuno was the best advisor that a student can get. He made an extraordinary effort to help me achieve my goals, and was always available to discuss and share ideas. He gave me the opportunity to be integrated in a large research group at an international level, which was a game changer for my perspectives.

I am also most grateful for having Sérgio Duarte and Carla Ferreira so close to me during the past few years. Sérgio and Carla had a fundamental role in giving elegance to this work. I would also like to highlight that I am grateful to Carla for trusting me and give me autonomy to work with her students in a later phase of my studies. Rodrigo Rodrigues was always very inspirational. He always contributed with a fresh view and profound understanding of the work. Last but not least, I would like to thank Marc Shapiro for the time we have spent discussing ideas with me, at a very early stage of my studies. I have never seen anyone else that puts so much effort in understanding people's ideas — that is a gift.

I am also thankful to José Orlando Pereira and Luis Caires for dedicating their time to give feedback and prepare me for the thesis defense. They had very insightful comments that helped me see my work in a wider spectrum. Likewise, I appreciate the time and availability of the jury to participate in the defense.

On a more personal note, I am grateful for establishing bonds with my office colleagues, whom I now call friends. I am very pleased to have met so many extraordinary people along the way and learn from other cultures. The following is a shortlist of people's names that I met during this period and will remain forever in my heart: Albert Linde, Alejandro Tomsic, Annette Bienusa, Bernardo Ferreira, Carlos Baquero, Cheng Li, Daniel Porto, David Navalho, Deepthi Devaki, Filipe Freitas, João Leitão, João Lourenço, João Silva, Jordi Martori, Manuel Bravo, Marek Zawirski, Ricardo Dias, Russell Brown, Tiago Vale, Zhongmiao Li.

None of this would be possible without my parents Ana and Fernando Sousa, that gave me everything so that I could reach this point in my life. It is impossible to retrieve or compensate for what they did, I can only aspire to one day be able to do for my children

what they have done for me. At last, a word for Adelina, Angela, Bruno and my closest friends: thank you for the patience, support and bringing joy to my life.

At last, I would like to acknowledge the institutions that supported my research: Departamento de Informática of the Faculdade de Ciências e Tecnologia of the Universidade NOVA de Lisboa (DI-FCT-UNL) and the NOVA Laboratory of Computer Science and Informatics (NOVA LINCS) for hosting me and providing all the equipment and space that I required; Fundação para a Ciência e Tecnologia (FCT/MCTES) for awarding me a scholarship (SFRH/BD/87540/2012); and, finally, the research projects: EU FP7 SyncFree project (609551), SwiftComp project (PTDC/ EEI-SCR/ 1837/ 2012), Rodrigo Rodrigues' ERC Grant (307732)

Abstract

The Internet has enabled people from all around the globe to communicate with each other in a matter of milliseconds. This possibility has a great impact in the way we work, behave and communicate, while the full extent of possibilities are yet to be known. As we become more dependent of Internet services, the more important is to ensure that these systems operate correctly, with low latency and high availability for millions of clients scattered all around the globe.

To be able to provide service to a large number of clients, and low access latency for clients in different geographical locations, Internet services typically rely on geo-replicated storage systems. Replication comes with costs that may affect service quality. To propagate updates between replicas, systems either choose to lose consistency in favor of better availability and latency (weak consistency), or maintain consistency, but the system might become unavailable during partitioning (strong consistency).

In practice, many production systems rely on weak consistency storage systems to enhance user experience, overlooking that applications can become incorrect due to the weaker consistency assumptions. In this thesis, we study how to exploit application's semantics to build correct applications without affecting the availability and latency of operations.

We propose a new consistency model that breaks apart from traditional knowledge that applications consistency is dependent on coordinating the execution of operations across replicas. We show that it is possible to execute most operations with low latency and in an highly available way, while preserving application's correctness. Our approach consists in specifying the fundamental properties that define the correctness of applications, i.e. the application invariants, and identify and prevent concurrent executions that potentially can make the state of the database inconsistent, i.e. that may violate some invariant. We explore different, complementary, approaches to implement this model.

The Indigo approach consists in preventing conflicting operations from executing concurrently, by restricting the operations that each replica can execute at each moment to maintain application's correctness.

The IPA approach does not preclude the execution of any operation, ensuring high availability. To maintain application correctness, operations are modified to prevent invariant violations during replica reconciliation, or, if modifying operations provides an

unsatisfactory semantics, it is possible to correct any invariant violations before a client can read an inconsistent state, by executing compensations.

Evaluation shows that our approaches can ensure both low latency and high availability for most operations in common Internet application workloads, with small execution overhead in comparison to unmodified weak consistency systems, while enforcing application invariants, as in strong consistency systems.

Resumo

A Internet tornou possível que pessoas em todo o mundo possam comunicar entre si numa questão de milissegundos. Esta possibilidade tem um grande impacto na forma como as pessoas trabalham, se comportam e comunicam, sendo que o universo de possibilidade ainda não é totalmente conhecido. No entanto, à medida que nos tornamos mais dependentes de serviços hospedados na Internet, maior é a necessidade de garantir que estes sistemas operam corretamente, com baixa latência e elevada disponibilidade para milhões de pessoas em todo o mundo.

Para conseguir providenciar serviço a um número elevado de clientes, e minimizar a latência de acesso para clientes em diferentes localizações geográficas, os serviços de Internet tipicamente recorrem a armazenamento geo-replicado. A replicação de dados possui custos associados que afetam a qualidade dos serviços. Para propagar as atualizações entre réplicas, os sistemas têm que escolher entre providenciar baixa latência e alta disponibilidade para executar operações, perdendo garantias de consistência (consistência fraca), ou manter a consistência, mas pagar um custo maior em termos de latência e perder disponibilidade para executar operações (consistência forte).

Na prática, muitos dos sistemas em produção adotam consistência fraca para melhorar a experiencia de utilização, negligenciado potenciais anomalias que estes sistemas podem causar nas aplicações. Nesta tese, estudamos a exploração da semântica das aplicações para construir aplicações corretas, sem prejudicar a disponibilidade e latência das operações.

Nós propomos um novo modelo de consistência que se afasta da ideia de que a correção de um sistema dependente da execução coordenada das operações entre réplicas. Nós mostramos que a maioria das operações pode executar com baixa latência, e de uma forma altamente disponível, mantendo a correção das aplicações. A nossa aproximação consiste em identificar as propriedades fundamentais que definem a correção de uma aplicação, isto é, os invariantes aplicacionais, e, através disso, detetar e prevenir a execução concorrente de operações que potencialmente possam tornar o estado da base de dados inconsistente, i.e. que possam violar algum invariante. Nós exploramos diferentes abordagens para implementar este modelo de consistência.

A abordagem Indigo consiste em prevenir operações conflituosas de executar concorrentemente através da restrição de operações que podem executar em cada replica, em

cada momento. Em comparação com soluções que combinação consistência forte e fraca, a nossa solução permite que algumas operações potencialmente conflituantes sem que isso afete a correção das aplicação.

A abordagem IPA não proibe a execução de nenhuma operação, garantindo alta disponibilidade. Para manter a correção das aplicações, as operações são modificadas para prevenir violações de invariantes durante a reconciliação de replicas, ou, se modificar as operações não produz uma semântica satisfatória, é possível corrigir o estado das aplicações antes que os clientes o possam ler, através da execução de compensações.

A avaliação experimental mostra que as abordagens estudadas permitem baixa latência e alta disponibilidade para a maioria das operações em aplicações para a Internet, com baixa penalização em comparação com sistemas de consistência fraca não modificados, enquanto fornecem garantias de consistência comparáveis às oferecidas por sistemas com consistência forte.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Over the last years, we have witnessed a change in the way people access the Internet. The personal computer comes in many sizes and shapes, allowing people to be connected to the Internet at all times. Nowadays, the Internet has no borders, people from any part of the globe can reach others in a matter of milliseconds, which makes it a very attractive channel for making business.

Companies have recognized the business potential of Internet services early. From e-commerce services, to advertisement and social media platforms, companies operating over the Internet span a wide range of industries. New services go live everyday with great impact on people's lives and routines.

In a very competitive market, companies need to provide services that are resilient and that can ensure quality of service to a large number of users. This is difficult for small companies that have limited resources to spend on infrastructure, but also for big companies that have to manage large and complex infrastructures. For these reason, more and more companies are offloading the management of infrastructures to the Cloud, which allows them to control the size of the infrastructure dynamically, in a cost-effective way [69, 70].

1.1 Context

At the core of Internet services are storage systems that manage and store data for applications. These systems need to be scalable, to ensure quality of service to a possibly very large number of users, and resilient to failures, to keep services operating even when nodes in the infrastructure fail or become partitioned. In this thesis we examine the design of large scale distributed systems, particularly those that use replication [29, 35, 40, 73].

Replication consists in maintaining multiple copies of data and applications logic, at different machines, for redundancy. It arguably helps improving performance and fault-tolerance of services, because it allows resources to be accessed from different endpoints. It can also help to reduce the latency for processing client requests, by forwarding them to a near-by replica, when available.

For companies that have users scattered across the globe, it is common to deploy replicas of data in strategic locations (geo-replication [35, 83, 130]) to reduce the distance between clients and services, helping to improve the overall perceived latency. Studies have shown that a slight increase in latency affects the user experience, with potential impact on the revenue of companies [42, 54, 91, 111].

To keep the state of replicas fresh, replicas need to exchange the updates executed in each of them. One way to do this, is to coordinate the execution of each operation across replicas, in order to keep their state synchronized at all times, or forward all updates to a single server that sequences the updates. This replication model is know as strong consistency [23, 24, 35, 83]. Despite the benefit of offering transparent replication, coordinating the execution of operations this way raises a number of problems. First, scaling systems that use strong consistency is difficult, because as more replicas are added to the system, more messages have to be exchanged to execute operations [52, 81]. Second, the execution of operations might be dependent on the availability of remote replicas, which might prevent the system from making progress if some replicas are unavailable. Finally, when replicas are far apart, the latency for executing operations might be too high. For these reasons, strong consistency is not adequate to be used over the wide area, and typical deployments are restricted to single data-centers, where infrastructure is more reliable and the latency is lower [43, 96].

An alternative way of executing operations is to first execute them at the replicas that receive the requests, and propagate the effects that they produce to other replicas asynchronously. This method is know as weak consistency and does not suffer from the limitations of strong consistency: systems can scale by adding more replicas, because operations are processed independently; ensures high availability, because as long as there is a reachable replica, the system remains live; and the latency for executing operations depends only on the distance between the client and the replica that processes the request (and the load on that replica). The downside of weak consistency is that the state of replicas diverge when updates are processed concurrently, because they are not immediately applied at all replicas. This makes programming systems that use weak consistency much more difficult than systems that use strong consistency, because the programmer has to account for concurrency anomalies.

In recent years, infrastructures that use weak consistency have emerged as the preferred choice for implementing large services, such as Amazon Marketplace, Twitter, or Facebook, since they offer a viable solution for providing low latency for clients that are scattered over wide areas. Many systems support weak consistency [22, 40, 78, 86], yet the difficulty of writing applications for these systems remains an open problem [11, 13,

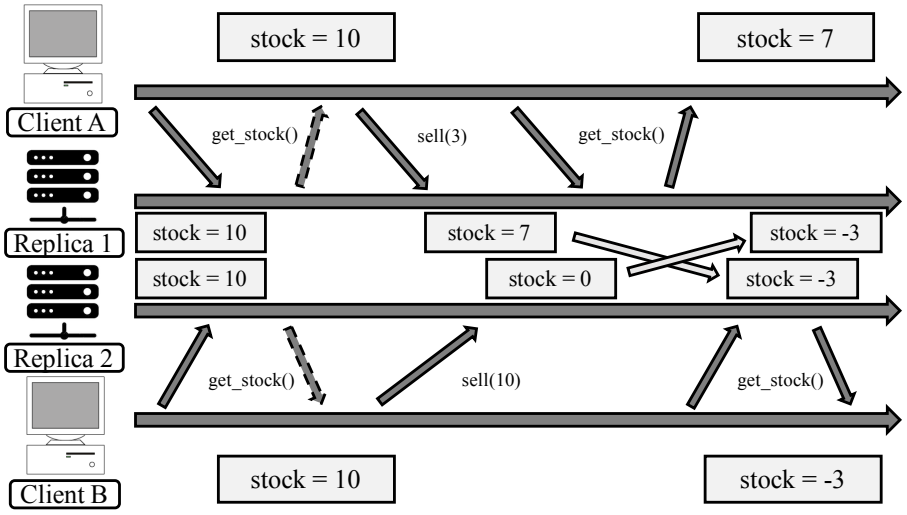


Figure 1.1: Two operations to buy an item execute concurrently at replicas *A* and *B*. The effects of the operations are propagated asynchronously to remote replicas, leading to a negative stock. This anomlay is not allowed under strong consistency.

58, 83, 112].

1.2 Thesis problem and statement

The difficulty of writing applications on top of weak consistency comes from the fact that concurrent operations might interfere with each other in ways that programmers cannot anticipate. For instance, consider an online store that tracks the availability of each item using a weakly consistent data store. Consider that the business logic of this service has a rule that says that the stock of items cannot become negative, i.e. the store does not allow overselling. To enforce this property, whenever an operation to sell an item is requested, the replica that receives the operation must check that the current stock is sufficient to process the operation, and only if it is, it processes the request. If two replicas execute this logic and do not coordinate the execution of the operations, they will not observe the effects produced by each other, allowing both of them to sell the last available units of some item concurrently, breaking the business constraint, as depicted in figure 1.1. This problem would never occur under strong consistency, as synchronizing the execution of operations would force the second operation to fail due to insufficient stock.

In one hand, we want to avoid coordinating the execution of operations to provide low latency and high availability and, in the other hand, we need coordination to ensure that applications are consistent at all times. The CAP theorem [26, 27] states that in a system prone to partitioning, it is impossible to ensure availability and consistency at all times. The intuition is that for enforcing consistency replicas must contact their remote counterparts, and for achieving availability replicas cannot depend on that. This result has driven the research of distributed systems in the past few years with different designs

exploring the two sides of the spectrum [11, 33, 40, 83, 86, 117, 130].

A prominent approach to this issue is to combine strong and weak consistency. The idea is to leverage the benefits of both approaches by using strong consistency for ensuring application correctness properties that cannot be maintained under weak consistency, and use weak consistency when the execution of operations is safe, to provide low latency and availability for those operations. To choose when to use one consistency model or the other, programmers need to identify which operations can be problematic, which is a difficult and error prone task. Many works have tried to automatize this process [7, 84, 110] to reduce the burden for the programmer, by leveraging the semantics of operations.

Separating operations by the consistency model that is most adequate in each case helps to achieve a good balance between strong consistency and weak consistency, however the overall latency and availability of the application might still be compromised if the number of operations that require strong consistency is high. In fact, using strong consistency for executing potentially unsafe operations might be inefficient in many cases, as all potentially conflicting operation executions have to be coordinated across replicas, even if some executions would not make the application state invalid. In the warehouse example, if the total number of units of an items that are sold concurrently at different replicas does not exceed the number of items available, the state of the application would remain valid even if these operations execute without coordination.

In this thesis, we question the premise that coordination is necessary for enforcing the correctness of applications, and that it always has an high penalty in the latency and availability of operations. Our hypothesis is that it is possible to leverage the semantics of applications to ensure the correctness of applications more efficiently. We show the validity of our hypothesis in two ways. First, we show that application's semantics allows us to reduce the number of cases where coordination is necessary to maintain correctness. We show that in certain cases it is even possible to avoid it completely. Second, we show that when coordination is necessary, in many cases operations can execute with low latency and in an highly available way, by moving the necessary coordination outside the critical path of execution, to avoid its costs during operations execution.

The work presented in this dissertation allows us to make the following statement.

Thesis statement: It is possible to leverage the semantics of operations to avoid coordination in a safe way, maintaining application's correctness and providing low latency and high availability in the general case. When coordination is unavoidable to maintain correctness, it is still possible to mitigate its cost for most operations execution.

An *application invariant* is a constraint defined over the state of the application that restricts the set of valid states of the application. The constraint on the items stock being non-negative is an example of an application invariant. While existing weakly consistent stores ensure state-convergence, they do not guarantee that the invariants of

the application are preserved at all times. On the other hand, strong consistency can be used to enforce invariants, by serializing the execution of operations, but performs badly.

In this thesis we propose an alternative approach for ensuring application correctness that departs from traditional approaches. Our approach follows the insight that the execution order of operations should be detached from application's correctness. We propose a new consistency model, called *explicit consistency*, that characterizes correctness as properties that the system must enforce. These properties are defined as a set of invariants over the system state. This simple, yet powerful, model allows programmers to specify the exact condition under which applications are correct, without implying any assumptions over the execution order of operations across replicas. While it is very desirable to ensure correctness under such simple model, developing applications under explicit consistency, without any guidance, could be very difficult.

We show that by providing the appropriate tools to programmers, it can become easier to implement applications under explicit consistency. Our tools provide a static analysis that is used to identify potential invariant violations in applications and ways to prevent those violations during application execution.

To make the approach easier to implement, we propose a 3-step methodology for developing applications under the explicit consistency model:

- Programmers specify the application invariants and operations effects;
- A static analysis identifies the pairs of operations that might violate any of the invariants defined in the previous step, and suggests modifications to the specifications to solve the identified conflicts.
- Programmers must modify the application code to implement the proposed specification.

We explore two complementary approaches for enforcing invariants in applications. The first approach consists in restricting concurrency by making replicas agree on which operations each replica can execute concurrently without requiring coordination, a technique know as reservations [21, 93, 104, 115]. On top of that, to reduce the impact that enforcing an agreement between replicas has in applications, we propose moving this step outside the critical path of operation's execution. This way, whenever a replica has permission to execute some operation, it can safely execute the operation locally, because remote replicas have agreed to not produce any effects that could result in an invariant violation. In the background, replicas proactively try to obtain permissions from remote replicas to ensure that sufficient permissions are available when executing operations. In the warehouse example, replicas agree on the number of items that each one might sell and can execute those operations safely without coordination, as long as they do not exceed their local budget.

The second approach consists in modifying the effects of conflicting operations to ensure that they can always be executed and propagated to any replica without affecting

correctness. The challenge with this approach is to ensure that the modifications that are necessary to prevent invariant violations still provide a reasonable semantics for the application. We show that we are able to provide alternative specifications with reasonable semantics for many invariants common in applications.

Our work provides a new insight over the trade-off between availability and consistency that helps to understand better the landscape of design choices for developing geo-replicated applications. Our work is the first to propose the utilization of invariants as a form of specifying the correctness of applications that run in geo-replicated settings. This approach opens path for addressing the problem of ensuring application consistency in novel ways. We propose a systematic approach for enforcing application correctness while minimizing the costs of coordination by combining reservations techniques and by modifying operations effects.

The reservation mechanisms that we propose are the first set of reservation mechanisms that can be used to avoid conflicting executions efficiently in geo-replicated settings. While we were inspired by classical reservation techniques, and follow the principles of coordination avoidance that has been explored by many authors [83, 84, 110, 117], our work improves the state of the art in a number of ways, making it possible to implement applications that provide low latency and high availability in geo-replicated settings. In the other hand, the techniques for modifying operations are unprecedented. We show, for the first time, how to maintain many classes of invariants under weak consistency without using coordination. While this solution cannot be used for all classes of invariants, it provides reasonable semantics for many classes of invariants that are common in applications. Both approach can be combined to avoid the pitfalls of each solution.

1.3 Contributions

The main contributions of this thesis are the following:

Explicit Consistency [16–18] We propose a new consistency model, the first of its kind, that captures the idea that the consistency of an application should be modeled after the invariant of the application, rather than by enforcing a particular order of operations execution. The key idea is that programmers should specify the invariants that must hold at all times and the system must enforce these invariants while minimizing the use of coordination. We have developed a set of tools that can help programmers to achieve this. Explicit consistency can be implemented by following the methodology described before and adopting the following complementary approaches to enforce application correctness.

Violation avoidance approach (Indigo [15, 16]) We propose to avoid invariant violations by restricting concurrency through the use of reservations [21, 93, 104]. A reservation is a mechanism that allows a replica to obtain rights to execute an operation

beforehand. If a replicas does not have enough permissions for executing an operation it must abort, or fetch rights from remote replicas at that moment, penalizing latency and availability. Although the idea of reservations has been proposed before, we are the first to adapt this idea to be used in a setting that combines weak consistency and geo-replication, providing a comprehensive set of reservations that can be used to ensure low latency and high availability. We first proposed the Bounded Counter, a data-type that can enforce numeric invariants without coordination for most operation executions. This approach allows us to cover a number of invariants common in applications, but has limited support for invariants that span multiple objects. We generalized the approach in Indigo, a system that provides explicit consistency by avoiding conflicting operations from executing concurrently.

Invariant preservation approach (IPA [19]) We propose IPA, a novel approach for preserving application invariants that completely removes coordination from operations execution. Instead of trying to introduce sporadic coordination to avoid invariant violations, we modify the effects of operations to allow them to execute concurrently with the guarantee that when replicas reconcile their state, the invariants of the application will always be preserved. This novel approach, advances the state of the art by providing a solution for maintaining certain classes of invariants under weak consistency, which have been deemed impossible before. Although modifying operations cannot preserve all classes of invariants, or in certain cases it is preferable to use coordination to attain a richer semantics, we show that we are capable of preserving a wide range of invariants that are common in applications without sacrificing latency and availability. We have developed a tool that can propose these modifications automatically.

Platform support and data types Indigo and IPA implementations heavily rely on convergent data types. In Indigo, we propose convergent reservation data types [21, 93, 104] that allow managing permissions to execute operations across replicas without coordination. In IPA we developed new data types with specialized convergence rules to support the transformation of operations.

We also proposed the design of a shim-layer for managing reservation data types on top of existing key-value stores and implemented it in various systems.

Evaluation We implemented several applications and benchmarks to demonstrate the performance and practicability of our approaches. We also make a qualitative evaluation of the effort for using them in practice, and analyze the types of invariants that we can cover.

1.4 Organization

The contents of the thesis is organized in 8 chapters.

In Chapter 2 we introduce essential concepts of distributed systems, with emphasis in replication techniques.

Chapter 3 discusses related work on the topic of replication and invariant preservation. We analyze the range of solutions that explore the consistency spectrum, discussing the programming models offered by the different system.

In chapter 4 we describe our first take on building invariant-preserving applications. In that context we have proposed the Bounded Counter, a data type for enforcing numeric constraints on the form of $x \leq K$, which are common in many applications.

Chapter 5 introduces the explicit consistency model. We present an algorithm for detecting conflicting pairs of operations, prove its correctness, and discuss the implementation of a tool to help programmers to use the methodology. The analysis presented in this chapter is crucial for implementing the approaches presented in chapter 6 and 7.

In chapter 6 we describe Indigo, the violation avoidance approach. We describe the different data types proposed for fixing the different classes of invariants, the middleware that provides support for these data types on top of existing Key-Value stores, and the evaluation of the approach.

In chapter 7 we describe IPA, the invariant preservation approach. We present the algorithm for modifying specifications, and the new data type semantics that are required for implementing them. We discuss the classes of invariants that are supported by the approach, and evaluation results.

In chapter 8 we conclude with final remarks.

Chapter 2

Background on Internet Services

A distributed system is any group of processes running in one or multiple machines connected through a network, working together to achieve some common goal. The goal might be to provide a service, produce some computation, or any unit of work that might be accomplished through the collaboration of processes.

Building scalable distributed systems poses significant challenges to programmers: decoupling service components introduces overheads that might impact performance; machines can fail arbitrarily, leaving parts of the system inaccessible, or malfunctioning; and communication between processes over the network is prone to failures and delays.

In this chapter, we describe the client-server architecture, to give an overview of the underlying infrastructure of Internet services, and discuss basic concepts of replication, transactions processing and consistency models of replicated systems.

2.1 Client-Server architectures

A client-server architecture, at its essence, divides the structure of a service in client and server components. Servers are responsible for managing data and processing requests, while clients handle requests between end-users and servers, and present their results. Clients typically provide functionality to a unique, or a small number of users, while servers are responsible for answering requests from the clients. At a high-level, the quality of a service is evaluated by the properties that it exhibits when clients interacts with it:

- **Availability:** Availability can be measured as the percentage of time that a service is able produce responses within a limit of time. A service is said to be *highly available* if it respond within a small time-frame at all times.

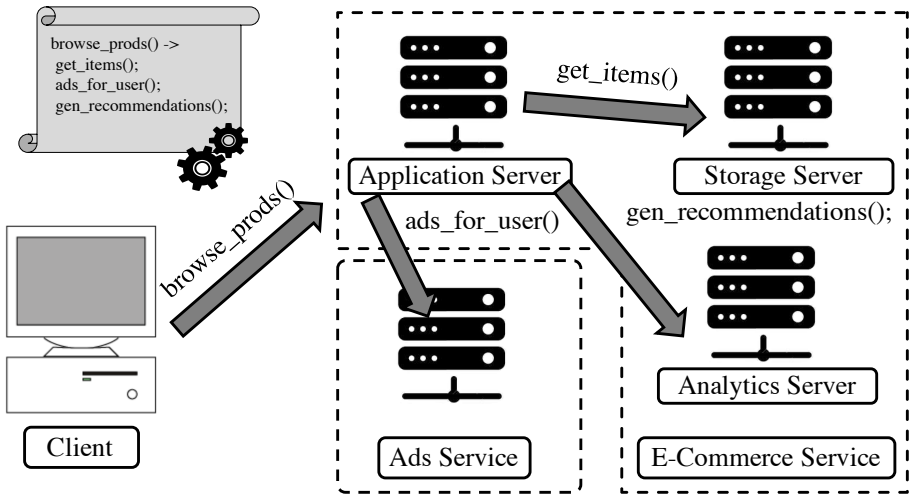


Figure 2.1: Diagram of interactions for processing a client request in a client-server architecture. The client sends a request for browsing items in a web store; the request is processed by an application server; product information is retrieved from storage server; data analytics is done by a dedicated server; and advertisement is handled by an external service.

- **Consistency:** A service is consistent if the execution of operations maintains the integrity constraints defined over the application state, i.e. that the state respects the invariants of the application as intended by the programmer.
- **Recency:** Updates must be made visible globally after execution. However, if a system is partitioned internally, or messages are delayed, updates might take time until they are made available to all clients.

Servers typically expose an interface of the application that clients can use to interact with the service. However, internally, requests might be processed by multiple servers that execute the logic of the application and/or store data. A tight-coupling between data processing and data storage reduces the latency for accessing data. However, this choice is less scalable because the access to data is conditioned by the load on the server for processing data, and vice-versa. For this reason, it is common to decouple the functionality of a server in two components, the *application server* and the *storage server*. Application servers deal with service logic and data processing, while storage servers are concerned with storing data and providing efficient access to it. This decoupling allows better scalability of individual components, as we discuss next.

Figure 2.1 describes the interactions between different servers of a complex service for processing a client request.

2.1.1 Providing services at global-scale

Companies want to provide their services for customers around the globe. To that end, they need to ensure good quality of service to millions of users, independently of their

location. This is particularly hard for services that operate on centralized infrastructures, because access to the service is only fast for users that are physically close to the servers, and because centralized services are prone to failures that might make the whole service inaccessible.

Application servers normally operate independently and are stateless, which makes it fairly easy to adapt to the load. When the system is under high load, this design allows to offload that load to other application servers that are added dynamically to the system, without interfering with the operation of other application servers.

Scaling the storage server is a more difficult task. A common approach for scaling the storage server is to partition the data across servers, to distribute the load [14, 29, 61, 118]. However, data is still subject to hot-spots and failures that might make the data inaccessible.

Scaling application servers and partitioning data efficiently allows internet services to achieve good performance. However, to ensure high availability and fault-tolerance, resources need to be replicated . In the next section we discuss how replication can enhance scalability, availability and fault-tolerance, by adding redundancy to the system.

2.2 Replication

Replication consists in maintaining multiple copies of data, and applications logic, that can be accessed and manipulated by clients, to improve the scalability and fault tolerance of a system. Some replication configurations scatter the replicas over different geographical areas (geo-replication) in order to reduce the latency between the end-users and resources [14, 35, 67]. Several replication schemes have been proposed that specify the topology of the replication infrastructure, determining how replicas connect with each other to process user requests and propagate updates [36].

In a replicated system, operations need to be executed among replicas in order to update their state. A consistency model defines the guarantees that the system provides regarding the execution of those operations. For instance, a consistency model might allow operations to execute in different orders across all replicas, or force them to execute in sequence. The programmer is responsible for ensuring that the system behaves correctly under these assumptions.

Ideally, the consistency model should be strong enough so that replication is transparent (strong consistency), i.e. the programmer cannot observe any side effects of replication. To attain such guarantees, normally, systems coordinate the execution of operations across replicas, or rely on a primary server to execute operations. However, these methods impacts scalability, availability and fault-tolerance, due to an higher cost for executing operations and the higher chance of failures, which are exacerbated by the physical distance between machines, in the case of geo-replication. Many existing systems, like Spanner [35], Farm [43], and others [3, 128], have been optimized to scale horizontally

under strong consistency, however they still exhibit high latency in the geo-replicated setting.

Some consistency models allow the state of replicas to diverge, in order to preserve performance and availability (weak consistency). In these systems, typically, an operation first executes in the replica that receives the request, and the effects of the operation are applied at a later time in the remaining replicas. This ensures low latency, independently of the location of replicas, because the replica that first executed the operation can reply immediately to the client, before propagating the effects to the other replicas. However, it is more difficult for programmers to use systems with weak consistency, because the data observed by clients might change between requests, when accessing different replicas, which make it more difficult to provide a good semantics for clients and ensure correctness [2, 23, 38, 58].

In this section, we analyze thoroughly different design aspects of replicated systems. We analyze different replication schemes, how to exchange the effects of operations that execute in different replicas, and discuss the benefits and drawbacks of different consistency models. For simplicity, in the remaining of this section we assume that operations are atomic and indivisible (transactions management is discussed in section 2.3).

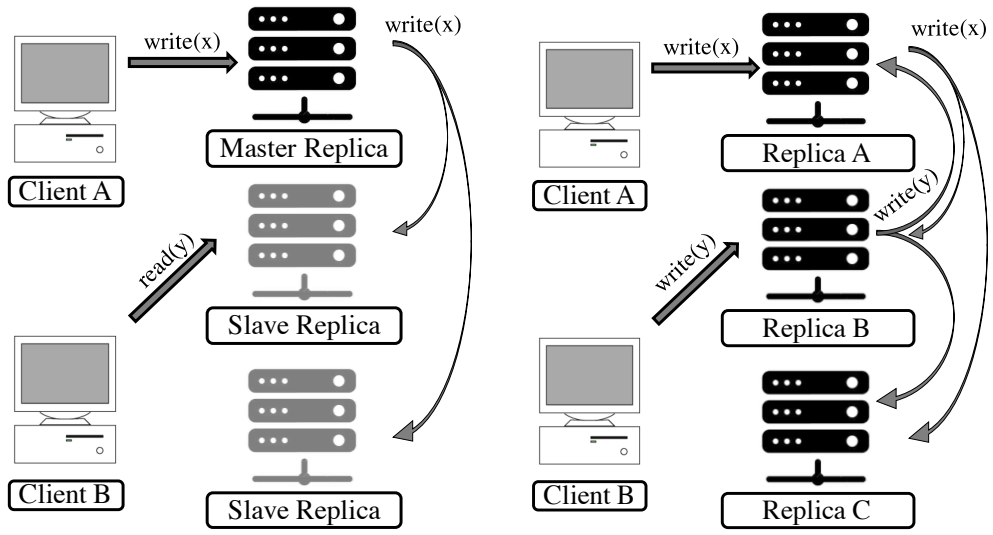
2.2.1 Replication schemes

In this section we analyze two classical replication schemes: *master-slave* and *multi-master* [38].

In a master-slave replication scheme, the master replica is responsible for processing all client requests and replicate the updates to the slave replicas [101, 119]. When the master fails, a slave replica is promoted to replace the master. Many systems that employ a master-slave replication scheme allow the slave replicas to process read-only operations to improve performance [34, 101]. This allows the system to scale well in applications that have a high rate of read-only operations, but the master remains a contention point for processing write requests.

To overcome this limitation of the master-slave scheme, the multi-master scheme allows any replica to process incoming update operations [73, 107]. However, if replicas coordinate with each other to execute every operation, the potential benefits of executing operations in any replica might be hindered by the cost of contacting other replicas. Choosing a weaker consistency model might prevent this issue, allowing replicas to exchange updates opportunistically, without affecting the performance of the system. The downside is that divergence might increase overtime, if replicas do not exchange updates frequently.

Figure 2.2 shows a schematic representation of the master-slave and multi-master replication schemes.



a Master-slave. Updates flow from Master to Slave.

b Multi-master. Updates propagated in every direction.

Figure 2.2: Replication schemes.

2.2.2 Replication unit

When designing a replicated system it is necessary to decide what data is propagated between replicas to update their state. This influences the number and size of messages that are exchanged, and the cost for processing messages at the origin and remote replicas.

Two alternative approaches that are widely used in practice consist in propagating the full state of objects, or just the effect of the operations that are executed [22, 40, 78, 130] (we discuss the implementation of data types that provide support for the two approaches in section 3.2.2).

In the first approach, operations modify the state of local objects, which are then propagated to remote replicas. In this approach, while objects are not propagated, they might accumulate multiple updates over time. When an object is propagated, the replica that receives the object simply stores the object if it does not exist, substitutes an existing value with the new one, loosing concurrent modification, or merges the local and the incoming values, to combine the modifications executed in both versions. To provide the latter semantics, the objects must have built-in support for reconciling different object versions.

When propagating updates, one way of executing operations is to separate their execution in two phases. First, in the origin replica, operations execute over the local state, determining the modifications (*effects*) that have to be produced in the object. Second, these changes are applied in the local replica and propagated to the other replicas of the system. This way, the replicas that receive the effects only need to apply the changes to the local version of the object, without re-executing the operation. Re-executing operations at remote replicas is complex, because it might be necessary to transform the operation

to reflect the original intention of clients [120].

Propagating objects state or updates might have advantages or disadvantages depending on the size of objects and the frequency of updates. When objects are large, but updates are small, is might be more advantageous to propagate updates individually, to save bandwidth. For instance, when a user posts in a social network, it might be better to propagate the new post than its entire feed. On the other hand, if updates are small, but very frequent, it might be more efficient to propagate the state of the object instead of individual updates. For instance, a counter object that stores the number of "*likes*" might be very small (e.g. the size of an integer), however, if a high number of operations is executed for that object, it might be more efficient to propagate the whole state of the object, instead of individual updates, as the state of the object might reflect multiple updates that were executed, reducing the number of messages that are exchanged, possibly with only a small increase in message size.

2.2.3 Propagation modes

Operations might be processed in coordination with remote replicas to ensure that the system state remains consistent at all times, or operations can be executed locally and propagated asynchronously to remote replicas, to enhance availability and fault-tolerance. We discuss the benefits and disadvantages of both approaches.

2.2.3.1 Coordinated execution

Coordinated execution consists in executing every operation synchronously with other replicas, typically, to enforce a total order of execution across them.

In a master-slave replication scheme, the master acts as a sequencer. Since all requests are forwarded to the master, operations first execute at the master, and then are applied in all slave replicas (or at least in a subset), in the same order, before replying to clients.

In a multi-master scheme, to enforce a total order of execution, replicas first have to agree on the order in which replicas process operations [25, 81]. In this case, the system might allow different operations to execute concurrently, as long as it is guaranteed that the observable state still conforms to a unique order of execution across replicas [24, 35].

Ensuring a total order of execution makes application development easier, because operations are applied against an unique view of the database state, free of concurrency conflicts. However, it also has a series of disadvantages. When executing an operation over different machines, the latency tends to be higher, particularly if nodes are physically disperse, which might increase the overall time for executing operations. Also, the coordination of replicas might require multiple steps of communication to ensure that every replica agrees on the execution order of operations [57, 81, 82, 94]. At last, if some nodes in the system are down, or inaccessible, the system might not be able to make progress until these nodes recover.

2.2.3.2 Asynchronous execution

In asynchronous execution, replicas are allowed to execute operations immediately and respond to clients before propagating the modifications to other replicas. In a master-slave scheme, where updates are only processed at the master, it is easy to ensure that slave replicas are consistent with the master as all updates are executed at a single node. However, in a multi-master setting it is more difficult to derive a total order of execution, thus systems typically allow the state of replicas to diverge over time. In these systems it is common to assume that the state of replicas can eventually converge to equivalent states, in case all nodes are connected and updates cease to arrive [22, 86, 112, 122, 130].

To ensure converge in the presence of conflicting updates (i.e. updates to the same object), normally the system has to provide some mechanism to reconcile replicas. The reconciliation mechanism depends on the type of data that is propagated between replicas. If replicas propagate the state of objects, the reconciliation might consist in taking all the modified objects since the last time two replicas synchronized, and reconcile the state of each modified object individually [22, 40]. If replicas propagates updates instead of objects, one way of implementing the reconciliation mechanism is to execute operations in the same order for each object, or in an order that is sufficient to ensure that the resulting states are equivalent [22, 120, 130].

Asynchronous execution typically exhibits low latency and fault-tolerance because operations only need to execute locally before replying to clients. The downside is that allowing divergence might make application development more difficult. For instance, when a replica fails and clients are handed-of to other replicas, the observed state might not contain all updates previously seen by the clients.

Figure 2.3 depicts a situation where a client executes an operation to set the value of a counter in one replica, but the next time it reads it, it accesses a different replica that does not reflect the effects of the previous operation executed by the client. Some consistency models, like causal consistency, constraint the set of replicas that a client can access to prevent these inconsistencies [34, 79, 86].

2.3 Transactions management

A transaction is a sequence of operations that appears to execute atomically to an external observer of the system at a single point in time. Transactions have an all or nothing semantics, meaning that the whole transaction executes successfully, or none of its effects are applied.

Systems that support transactions need to implement concurrency control to ensure that multiple operations execute atomically and in isolation from other concurrent transactions. Isolation from concurrent transactions is necessary to ensure that operations from different transactions that access and modify the same objects do not interfere with each other. Ideally, concurrency control ensures *ACID* properties for transactions [60]:

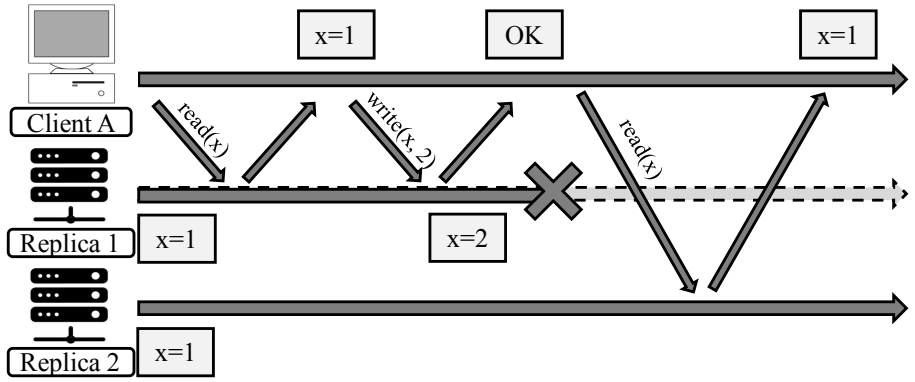


Figure 2.3: Client reads object x and writes $x = 2$ to replica A; Replica A accepts the write, replies success, and then fails, before writing the value to replica B. Client reads the value x from B and gets $x = 1$.

- **Atomicity:** All effects of a transaction are made visible or none;
- **Consistency:** The state of the database always transitions between correct states (i.e., preserving the integrity of data);
- **Isolation:** Two transactions that execute overlapping in time, do not observe effects of each other;
- **Durability:** When a transaction commits, its effects are permanent.

Transactions with ACID properties are said to be *Serializable*. This means that transactions appear to an external observer to execute one after the other. The runtime might allow executing transactions concurrently, but the system must ensure that it is possible to still derive a sequential execution order for those transactions [2].

Ensuring ACID properties for transactions requires a great deal of coordination. When systems are deployed in a reliable environment, like inside data-centers, the latency between nodes is low and failures are easily masked. This allows the development of specialized protocols that perform well and ensure ACID properties for executing transactions, like FaRM [43], or Spanner [35]. However, when data is stored over the wide-area, since the latency and the risk of failures is much higher, ensuring ACID properties for transactions imposes great overheads for coordinating the execution of operations across data centers [4, 35, 117].

Some system, like RAMP [12], SwiftCloud [130] and COPS [86], weaken the consistency and isolation criteria of storage systems to provide better availability and performance, while still preserving reasonable semantics for executing transactions (discussed in more detail in section 3.2.4).

2.4 Consistency models

Consistency models define the set of rules that a system must obey when executing operations over replicated data. The state of replicas is said consistent, according to some consistency model, if the execution of operations across replicas respect the rules specified by that model. While the state of replicas might be consistent according to some model, it does not prevent applications' state to become incorrect. The programmer is responsible for ensuring that applications are correct given guarantees defined by the model.

Consistency models can be broadly classified as strong and weak. We use this classification throughout this thesis to refer to consistency models that fall into the two categories.

Under *strong consistency*, the system appears to behave as if it was not replicated underneath. Strong consistency is typically achieved by coordinating the execution of operations across replicas, which provides only limited fault-tolerance and availability. ACID transactions fit into this model.

Under *weak consistency*, the system is allowed to expose different replica states to clients. Weak consistency is usually implemented using asynchronous execution, providing high availability and fault-tolerance, but allows concurrency conflicts.

In an ideal world, we would like geo-replicated systems to attain the guarantees offered by strong consistency models, and the performance and availability that weak consistency models allow. However, that is practically impossible due to the time messages take to be propagated from one location to another and due to the possibility of network partitioning. In practice, system designers have to take into account the characteristics of the deployment and the service when deciding which consistency model to use in their systems.

However, in many cases, enforcing strong consistency is just a way to make it simpler to handle concurrency in applications. Many applications do not require such strong consistency criteria, or, at least, they do not require it at all times. For example, when selling indistinct items, it is not necessary to enforce an unique order of operations, as long the items are not oversold. The system might sell items in any order across replicas, as long as there is enough stock, and only fallback to a sequential execution of operations when the stock is low to avoid overselling items [75]. Relaxing the order requirements can have a huge impact on applications performance, especially when replicas are far apart and latency for executing operations is high, as in geo-replicated scenarios. Some services also are more tolerant to the reorder of operations than others. For instance, in a social network it might not be important that the feed of publications appears in a different order to different clients. On the other hand, when managing an auction, it is important that bids are globally ordered.

Different consistency models explore the trade-offs between providing good consistency properties and ensuring good availability for different classes of applications. We

now describe the characteristics of some consistency models, from the strongest to the weakest.

- **Strict Serializability [43, 64]:** Any read operation for object x must read the most recent value of x. The most recent value of some object is the version generated by the last update that executed in any replica, at the exact moment the operation is issued. In this model operations execute in sequence, according to the time order in which they are issued. This model is expensive to implement in practice, as it requires that writes become atomically visible to all replicas at a single point in time.

- **Serializability [24]:** The time constraint of strict consistency is dropped, requiring only that operations execute in some sequential order. The order of operations of the same client must respect their relative order, and operations from different clients must also appear to executed according to an unique sequence, in all replicas. Operations are allowed to be reordered as long as it does not affect the values observed in each operations.

- **Causal Consistency [79, 86]:** Under causal consistency operations that potentially depend on others appear to all replicas in an order that preserves those dependencies. This means that if an operation b executed after some operation a, it will only be visible in some replica after operation a is also visible in that replica. If two operations do not depend on each other, we say they are concurrent and can be applied in any order. Causal consistency is very useful for programmers as it prevents many execution anomalies that can occur due to asynchronous propagation of operations (explained in more detail in section 3.2.4.2).

- **Eventual Consistency [40, 126]:** This is the weakest consistency model defined. It only requires that the database state converges, if updates cease all replicas. In this model operations can be propagated and executed in any order, as long as the system is able to achieve convergence.

2.5 Final remarks

In this chapter we have covered basic concepts of Internet services. We have focused on replication, a technique that we study in this thesis. We have discussed the architecture of replicated systems and how to execute operations across replicas. These techniques are fundamental to understand the design of the solutions that we describe next.

We have stressed the importance of ensuring high availability and low latency, and the difficulty of ensuring consistency in that setting. We described the properties of ACID transactions, which are convenient for programmers and discussed the difficulty of providing those guarantees in the wide area. We have presented different consistency

models that are commonly used in existing systems. In the next section, we discuss in more details the practical implications of consistency in applications.

Chapter 3

State of the Art

The landscape of research in distributed storage systems is greatly influenced by the trade-off between consistency and availability. In this chapter, we present different ways in which existing works try to address this trade-off.

In section 3.1 we present the CAP theorem, the theorem that states the impossibility of achieving consistency and availability at the same time in a distributed system. Following that result, many works have been proposed that try to address this pivotal trade-of of distributed system, either by providing scalable solution that try to ensure consistency or availability, and solutions that try to find a good compromise between both choices. In this chapter we review a selection of those works.

We start by discussing the design of weakly consistent stores in section 3.2. These databases provide low latency and high availability for services that work on a global scale. Despite that, these systems cannot be used to implement many applications correctly, due to the difficulty to enforce certain kinds of invariants.

Section 3.3 presents the line of work that combines weak and strong consistency. The objective is to use weak consistency whenever the execution of operations does not put consistency at risk, and fall back to strong consistency when it is necessary to ensure correctness.

It is not easy to use systems that combine different semantics for executing operations, thus, many existing systems try to mask the downsides of strong consistency to clients. We study those systems in section 3.4. The idea is to provide rich programming models for developers that allows them to make applications responsive, while transactions execute across replicas in the background.

Our work receives inspiration from works presented in this chapter. The contribution lies in the exploration of application's semantics to provide better consistency without sacrificing availability and latency of systems, attenuating the gap between consistency

and availability. Finally, We discuss how our work compares to the state of the art in section 3.5.

3.1 CAP Theorem

In 2000, at PODC, Eric Brewer gave a historic keynote about the design trade-offs that programmers face when building distributed systems. In this talk he proposed the CAP theorem [27] that says that a system can only ensure, at all times, two out of three desirable properties: consistency, availability and fault-tolerance.

In systems that run on a single-site, if there are no internal partitions, the system can ensure availability and consistency at the same time. In the wide-area, the common assumption is that the network might get partitioned or fail arbitrarily. In that case, if a client sends a request to a replica, either the system responds before receiving the responses from replicas in other partitions, loosing consistency, or the replica waits indefinitely for the responses, loosing availability, since the response might be delayed or never arrive.

The take away is that, in practice, when the system is in normal operation mode, it is able to ensure consistency and availability at the same time. But, when the system is partitioned, either it has to choose to preserve consistency or availability.

The theorem was later formalized by Gilbert and Lynch [53], where the authors prove the theorem in asynchronous and partially-synchronous systems.

In one hand we want services to be available, but, in the other hand, we need to maintain applications correctness. In large-scale systems it is common to favor availability (which also offers better latency) over consistency, to ensure that the system meets a response latency that is acceptable for users around the globe. This poses limitations to the class of invariants that can be maintained correctly, limiting the applications that can be run on top of these systems.

To find a compromise between availability and consistency, many systems opt for bringing the decision of ensuring availability or consistency to the programming level [34, 83, 117]. This way the programmer can choose which is best in each situation. However, this is cumbersome for programmers, because it makes more difficult to program applications and to ensure that they are correct.

In the next section we discuss the design of systems that only provide weak consistency, highlighting some of their limitations, and in the following section, we discuss how to use coordination efficiently to overcome them.

3.2 NoSQL databases

The Dynamo paper [40] proposed a new database architecture that breaks apart from traditional system designs. Dynamo is a distributed Key-Value store that favors availability and fault-tolerance over consistency, opening the path for NoSQL database systems.

Since its inceptions, many systems, including academic [6, 44, 86, 87, 108, 117, 130] and commercial [22, 34, 78] have implemented variants of this design. These databases are characterized by employing a very simple data model, in contrast to classical database systems [59], which allows them to be more scalable. In this section, we discuss the design of these systems.

3.2.1 Key-Value stores basics

Elasticity is fundamental for companies operate their business successfully, as many web workloads are prone to huge variations of load, thus these systems have to adapt dynamically to ensure good quality of service and minimizing costs (e.g. shopping seasons, viral news). Key-value stores typically provide elastic storage, allowing nodes to enter and leave the system dynamically, and adapt to the current load of the system without disruption of the service.

Key-value typically have a simple interface with get/put operations over binary objects. In some cases, they might provide support for more complex object types, instead of simple binary objects (e.g. Riak [22] ad SwiftCloud [130]). Underneath, the system is composed by a set of nodes that store the keys.

Topology The key-space of the store is distributed across the nodes that compose the system. To this end, a consistent-hashing function determines the node that is responsible for storing each key. Consistent hashing distinguishes from normal hash functions by allowing that only a fraction of the keys it maps need to be moved when nodes enter or leave the system.

The idea is that the range of the hash function is treated as a fixed circular space, where the largest value wraps around to the smallest. Each key is mapped to a particular point in this ring, and each node is assigned to a range of points of the ring, storing the content of keys that fall into its range. To access the value of a key, the runtime finds the (virtual) position of that key in the ring, and transverses the hash space to find the node that stores that key. That node is responsible for storing all keys that fit between him and the previous node in the ring. When a node enter the system it takes a place in the ring and the neighbor hands-off the corresponding keys. The symmetrical process occurs when a node leaves the systems orderly.

Replication To provide fault-tolerance, a replication scheme is implemented on top of the ring. Each value that is assigned to some partition is replicated over the next N partitions that are stored in different physical machines, in sequential order, where N is the replication factor.

For executing $put(k, v)$ and $get(k)$ operations, the programmer must set the number of replicas that the system has to contact before replying to the client [36, 52]. Reading and writing to a subset of the available replicas provides faster responses, but it might

allow the creation of multiple versions of objects, if updates are executed concurrently for the same object in different subsets of replicas. In Dynamo, if at any time, a node holds multiple versions of the same object, they are delivered to the client upon read, whom is responsible for solving the conflicts in some way. In the next section we describe a systematic approach for automatically handling conflicts in applications.

3.2.2 Conflict-free Replicated Data-Types

In weakly consistent systems, concurrent updates for the same objects have to be reconciled to ensure state convergence. Leaving the task of reconciling updates to the programmer makes applications development more difficult. Also, automatic strategies that pick only one version of each object have a bad semantics because it discards updates. An alternative solution, that provides a more useful semantics for programmers, would be to integrate the concurrent updates in a single object version systematically without losing any operation. For instance, in a replicated counter, if the counter is incremented concurrently, we would like to sum all increments, instead of choosing one of the values of the counter to prevail.

Conflict-free Replicated Data-types (CRDTs) [113] are a form of replicated data types [20, 109, 113] that can handle concurrent updates automatically, based on some well-defined convergence semantics. This abstraction is convenient for programmers that already have to implement applications using abstract data types.

A large number of CRDTs have been proposed, including versions of common data-types used in applications, like counters, registers, sets and maps [112]. Some existing key-value stores support CRDTs directly in the interface of key-value stores [22, 130], making easier to implement applications on top of weak consistency.

CRDTs can be implemented correctly without using coordination for executing operations, and support state and updates propagation for synchronizing the state of replicas. In the next section we discuss how to implement CRDTs using the two approaches.

3.2.3 Implementing CRDTs

A CRDT object has an internal state, the payload, and a defined set of operations to read and modify its state. In this section we describe how to implement CRDTs using state-based and operation-based approaches.

3.2.3.1 State-based

A join-semilattice is a partially ordered set $(S, \leq_l)$, equipped with a least upper bound function (LUB): given two elements a and b in the join-semilattice, the LUB of those elements, is another element in the join-semilattice, c, such that $a \leq_l c$ and $b \leq_l c$, according to $\leq_l$ and there is no other $d \leq_l c$ such that $a \leq_l d$ or $b \leq_l d$. In other words, the LUB function takes two objects of the same type and computes a new version of that object

that is the smallest value that is greater than the two. The merge operation is idempotent, associative and commutative.

A state-based CRDT is any structure that has the properties of a join-semilattice. Update operations always have to make the state of the object move upward in the partial order $\leq_l$, i.e., they are monotonic. To synchronize the state of state-based CRDTs , each data-type has a $merge(A, B)$ function that computes the LUB of two objects A and B of the same type. For instance, in a counter object, $merge(A, B)$ consists in summing the increments and subtracting the decrements executed in each object. It is desirable that the merge function always computes a new object state that contains the updates executed in each object, however it is possible to derive other merge semantics that respect the definition of the LUB.

To reduce the cost of propagating the whole state of the object each time replicas synchronize, delta CRDTs [5, 85] just propagate the mutations executed on the object since the last synchronization. Deltas can also be aggregated to save the overhead of transmitting multiple messages.

State-Based Counter CRDT

The increment-only counter is a counter, whose only update operation is to increase the value of the counter. The pseudo-code for the specification of this data type is given in algorithm 1. The payload of the object consists of a vector, with an entry for each entity that has modified the counter (for instance, each replica), initialized with value 0. The $increment(n)$ operation increases the vector entry corresponding to the entity that issued the request by n units. To compute the current value of the counter, $query()$ sums all entries of the vector, which totals to the number of increments executed in the counter. The $merge(A, B)$ operation takes all the increments executed in each replica by taking the maximum values of each entry of the vector. The proof that this data type is a CRDT is shown elsewhere [112].

Figure 3.1 presents a fragment of the join-semilattice of the increment-only counter, using a Hasse Diagram. Each element in the lattice corresponds to a possible values of the object. Edges represent state transitions, which are obtained either by executing *increment*() or *merge*() requests. Elements that have a single incoming arrow represent states obtained from executing *increment*() operations, while elements with two incoming edges represent states obtained by executing the *merge*() operation. Elements in higher levels are greater than elements in lower levels, according to the partial order, and elements at the same level are not ordered in respect to each other.

To extend this counter design to support decrement operations, we can combine two increment-only counters, and use them to store increment and decrement operations separately. Algorithm 2 presents the specification of the Positive-Negative counter (PN-Counter) in pseudo-code. The internal state of the object are two separate vectors that store increments, P, and decrements, N. Operation $increment(n)$ increments P, and

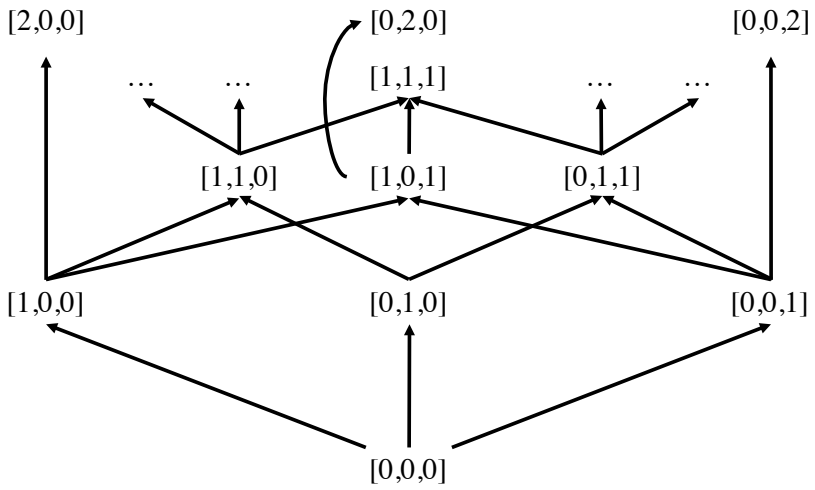


Figure 3.1: Fragment of the lattice of the increment-only counter CRDT. Edges represent state transitions through increment or merge operations. Elements are ordered according to $\leq_{i-counter}$, in crescent order bottom-up.

Algorithm 1 State-based Increment-only Counter.

```
1: payload integer[n] P                          ▷ N: number of replicas
2:     initial [0,0,...,0]
3: update increment (integer n)
4:     id = repId()                              ▷ id: id of the replica that issues the request
5:     P[id] = P[id] + n
6: query value () : integer v
7:     v = Σ_{i∈Ids} P[i]
8: update merge (S)
9:     P[i] = max(P[i], S.P[i]), ∀i ∈ Ids
```

operation $decrement(n)$ increments N. To get the current value of the counter, the $query()$ operation sums the entries of each vector and returns their difference.

3.2.3.2 Operation-Based

In operation-based CRDTs, replicas synchronize their state by propagating the effects of the operations executed in each replica. The execution of operations is separated in two phases: prepare, and downstream. The prepare phase, which is only executed in the local replica that requested the operation, determines the set of effects produced by the request. Those effects are applied in the downstream phase, which is executed in all replicas.

In this case, we assume that the channel implements reliable causal-order broadcast, i.e. all messages are delivered exactly-once to all participants, in causal order. Since the model allows operations to execute concurrently at different replicas, concurrent downstream phases must be commutative with each other to ensure state-convergence. This design has been been proven a CRDT elsewhere [112].

Algorithm 2 State-based PN-Counter.

```
1:  payload integer[n] P, integer[n] N                ▷ N: number of replicas.
2:      initial [0,0,...,0], [0,0,...,0]
3:  update increment (integer n)
4:      id = repId()
5:      P[id] = P[id] + n
6:  update decrement (integer n)
7:      id = repId()
8:      N[id] = N[id] + n
9:  query value () : integer v
10:     v = Σ_{i∈Ids} P[i] − Σ_{i∈Ids} N[i]
11: update merge (S)
12:     P[i] = max(P[i], S.P[i]), ∀i ∈ Ids
13:     N[i] = max(N[i], S.N[i]), ∀i ∈ Ids
```

Algorithm 3 Operation-based Counter.

```
1:  payload integer i
2:      initial 0
3:  update increment (integer n)
4:      downstream()
5:          i := i + n
6:  update decrement (integer n)
7:      downstream()
8:          i := i − n
9:  query value () : integer v
10:     v = i
```

Operation-Based Counter CRDT

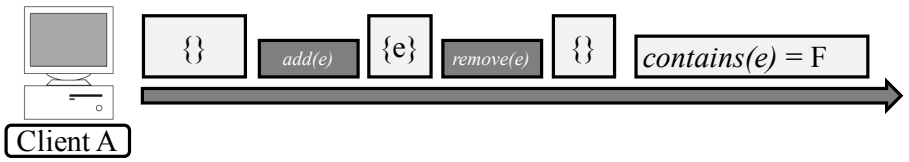
Algorithm 3 presents the specification of an operation-based counter CRDT in pseudo-code. The payload of the object is a single scalar. Since the communication layer guarantees that messages are delivered only once to each replica and all operations are commutative, the data-type simply has to apply each incoming update in causal order. In comparison to the state-based approach, less meta-data is required to implement this data-type, but assumes more guarantees from the underlying system.

3.2.3.3 Convergence policies

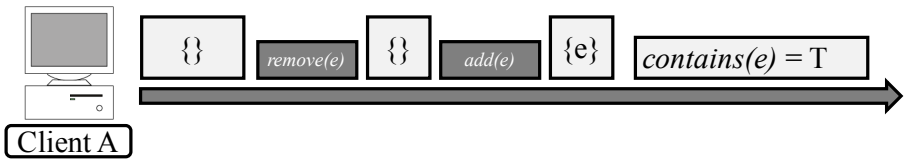
In some data type designs, some operations are not commutative. For instance, in a set data type, the semantics of executing *add*(*e*) followed by *remove*(*e*) is different from the execution of *remove*(*e*) followed by *add*(*e*), as depicted in figure 3.2.

To make these CRDT implementations of these data types we need to: (i) define what should be the resulting state of operations in the presence of concurrent updates; (ii) implement a design that achieves that goal.

To decide what should be the outcome of concurrent operations execution, we need to take the intention of clients in consideration. Consider the execution in figure 3.3a, where the set has an element *e* and *ClientA* adds the element again to the set, while another *ClientB* removes it. Clearly both operations are not related but interfere with



a Execute *add*(*e*) followed by *remove*(*e*).



b Execute *remove*(*e*) followed by *add*(*e*).

Figure 3.2: Semantics of the set data type.

each other, thus we need to define a convergence semantics for that execution that defines the resulting state of applying both operations. One possible outcome is to maintain the element in the set, giving "*priority*" to the *add*(*e*) operation. The *Add-wins* strategy implements this behavior, ensuring that element *e* is in the set even if a concurrent remove operation removes a previously seen instance of that element.

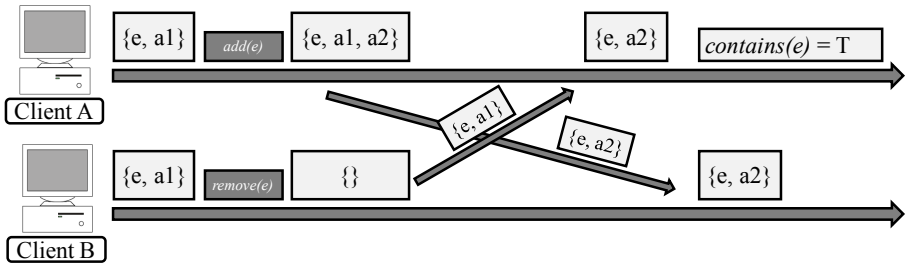
The op-based specification of the Observed-Removed Set (or simply *Add-wins* Set) is presented in algorithm 4.

The payload of the object consists of a set of pairs (*element*, *uid*). The *add*(*e*) operation associates a tag (unique global id) to the element that is being added to the set. On *remove*(*e*), the tags visible at the replica that executes the operation are removed from the set, and propagated for removal in the remaining replicas. If an *add*(*e*) operation executes concurrently for the same element, the *remove*(*e*) has no effect over that operation because the tag generated by the *add*(*e*) is not visible to the remove operation. The *contains*(*e*) operation returns true it there is at least a unique tag for element *e*. The example of an execution is shown in figure 3.3a.

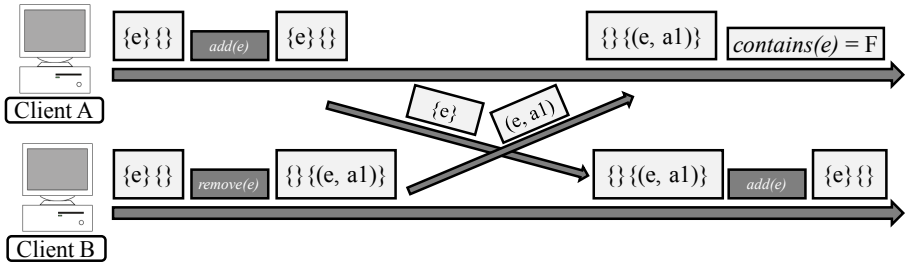
Conversely, the *Rem-wins* policy, ensures that if some operation removes an element *e* it has precedence over any concurrent *add*(*e*). Instead of generating tags when adding elements, tags are associated to elements on remove. When a tag exists for some element *e*, the element is not visible in the set until an *add*(*e*) operation that observes that tag executes and removes all existing tags. The op-based specification of the *Rem-wins* set is given in algorithm 5 and the example execution is shown in figure 3.3b.

3.2.3.4 State convergence vs. invariant preservation

CRDTs abstract from programmers the difficulty of ensuring state convergence. However, in many cases, ensuring that replicas eventually converge is not a sufficient condition for ensuring application correctness.



a *Add-wins* semantics.



b *Rem-wins* semantics.

Figure 3.3: Examples of concurrent semantics of the set data type.

Algorithm 4 Operation-based Add-Wins Set.

```
1:  payload Set S
2:      initial ∅                          ▷ Set of pairs Set of pairs (element, unique_id)
3:  query contains (element e) : boolean b
4:      b = (∃_uid : (e, uid) ∈ S)
5:  update add (element e)
6:      prepare(e)
7:          uid = unique()                 ▷ returns a global unique identifier.
8:      downstream(e, uid)
9:          S := S ∪ {(e, uid)}
10: update remove (element e)
11:     prepare(e)
12:         if contains(e) then            ▷ Pre-condition.
13:             R = {(e, uid)|∃_uid : (e, uid) ∈ S}
14:     downstream(R)
15:         S := S \ R                     ▷ Removes pairs identified at source.
```

For example, consider an application that stores information about available items in a warehouse and that it allows selling the available items at different replicas of the system without coordination. To ensure that the stock information eventually converges, we can store the stock of each item using a counter CRDT. The problem is that when a replica issues an operation to decrement the stock, it will only check the local value of the object, while other replicas might have already modified that value concurrently. This might allow the value of the stock to become negative after applying all concurrent operations, even if the stock was positive in each replica when the operations first executed.

Despite the ability to enforce database state convergence, application's integrity constraints are not necessarily enforced by the design of the data type. In chapter 4 we

Algorithm 5 Operation-based Remove-Wins Set.

```
1:  payload Set E, R
2:      initial ∅,∅                          ▷ E Set of element, R Set of pairs (element, unique_id)
3:  query contains (element e) : boolean b
4:      b = (e ∈ E) ∧ not(∃_uid : (e, uid) ∈ R)
5:  update add (element e)
6:      prepare(e)
7:          D = {(e, uid)|∃_uid : (e, uid) ∈ R}
8:      downstream(e, D)
9:          E := E ∪ {e}
10:         R := R \ D                       ▷ Removes all visible uids.
11: update remove (element e)
12:     prepare(e)
13:         if contains(e) then              ▷ Pre-condition.
14:             uid = unique()
15:     downstream(e, uid)
16:         R := R ∪ {(e, uid)}
```

introduce a new CRDT design that prevents invariant violations like the one described. The idea is to enforce a local check that prevents the execution of the operation if the replica cannot guarantee that the global invariant is maintained.

3.2.4 Rich semantics for weak consistency systems

As we have pointed out, key-value stores, forfeit consistency to ensure better availability and performance. In this section we present extensions to eventually consistent systems that provide better consistency semantics, while making reasonable trade-offs of performance and availability.

3.2.4.1 Session Guarantees

In storage system that allow clients to read and write to multiple replicas, it might happen that different subsets of replicas process different requests from the same client. This allows that clients interacting with different replicas of the system over time observe values of objects that do not reflect previous interactions. For instance, a customer changes his password, by issuing a write operation in some replica. Next, the user tries to log-in, but the request is processed by a different replica that has not seen the previous update. The user types the new password but the log-in fails because the new value has not yet been applied in that replica.

Session guarantees [123] define a set of guarantees that the state of a replicated system must exhibit to a client that interacts with it over a continuous session. Various guarantees are proposed:

Read your writes: ensures that a client can only access a version that contains the previous writes executed by him, in that session.

Monotonic reads: ensures that if a client reads some set of updates, then subsequent reads must always observe states that contain at least that set of updates.

Write follow reads: ensures that if a client writes some object version a after reading some set of object versions B, then any accessed state that contains a must also contain B.

Monotonic Writes: ensures that a writes from the same client preserve their relative order across replicas.

In the previous example, the *read your writes* guarantee is sufficient to solve the identified problem as it would prevent the client from accessing any replica that has not seen the operation to change the password.

Brzezinski et. al have shown that the combination of all session guarantees ensures causal consistency [28].

3.2.4.2 Causal Consistency

Causal consistency strengthens eventual consistency by enforcing a partial order among operations that is compatible with the happens-before rule [79]. In practice, this means that if some operation op_b executes in a state S that reflects the updates of an operation op_a, then op_b is only visible in states that reflect the effects of op_a. This is a necessary condition to preserve the intention of the user when he executes multiple operations that are related with each other but might be processed in different order in different replicas. For example, consider a social network that stores photos of users and permissions to access those photos in different objects (possibly in different machines). Some user wants to share some photos of a party, but he is worried that his employer sees the photos online. To prevent that, he reduces the access level of the photos to exclude the employer, and afterwards upload the photos. The second operation, the operation to upload the photos, should only be visible at some replica if the previous operation has also been applied, to ensure the user's intention that the uploaded photos should not be visible to the employer is maintained. If, during propagation, the effect of the upload operation is made visible before changing permissions in some replica, the employer might be able to see the photos. Causal consistency precludes that by ensuring that the effects of the upload operation are only made visible in some replica after the operation to change permissions is applied in that replica.

Causal consistency has been shown to be the strongest weak consistency model that can be implemented in an always-available fashion [88], i.e. that the system remains available under partitioning.

The definition of causal consistency does not enforce state convergence, it only requires that operations respect the happens-before rule. It is common that systems that enforce causal consistency also require state-convergence. This model is called *causal+ consistency* and is widely implemented in practice [4, 6, 9, 45, 86, 87, 130].

The downside of providing causal consistency is that additional meta-data is required to to track updates dependencies. The dependency graph of each object version contains all object versions that were visible to the operation that that created that object version. This graph might be very long, and incur in non negligible processing overheads to determine if a certain object version is visible.

Causality tracking

Since it is expensive to check the dependencies graph for each object systems optimize the process in different ways [6, 44, 45, 86, 87, 92, 130]. We describe three systems that track dependencies differently and highlight the trade-offs in causality tracking.

In COPS [86], a remote operation is only applied in some replica after all object versions that the operation depends on have also been applied in that replica. Applying operations in this order ensures that if some dependency is satisfied, then all previous dependencies have also been satisfied. This way, the system does not have to record dependencies that are more than 1-hop away in the dependency graph (i.e. they are automatically satisfied).

In SwiftCloud [130], causality tracking is done by means of version vectors [99]. A version vector is used in each replica to summarize the operations that the replica has seen from other replicas. Each entry in the vector stores the number of the most recent update received from every other replica (operations are applied in FIFO order to ensure that the value of the entries is strictly monotonic). Each object version has an associated version vector that identifies the dependencies of that object. Objects can be propagated between replicas asynchronously, but they can only be made visible at remote replicas after their dependencies are satisfied. To decide if an object version can be made visible, the replica checks if its version vector is greater or equal to the version vector of the object. In comparison to COPS, In Swiftcloud the meta-data size depends on the number of replicas in the system, while in COPS it depends on the the number of dependencies for each object.

GentleRain [45] reduces the size of the dependencies of each object to a single scalar. This might slow down the visibility of updates because, under this approach, it is impossible to distinguish if a remote update that has not been seen is a dependency for some local object or not, therefore the system must wait for all potential dependencies to arrive.

The protocols described make different trade-offs between meta-data size and visibility latency, highlighting a trade-off in causality tracking: as the more compressed are dependencies, the faster it might be to transmit them, but it also increases the chance of waiting for false dependencies.

3.2.4.3 Transactions in weak consistency systems

ACID transactions require synchronous commit protocols that might execute multiple steps before finishing a transaction and making it durable. These protocols have high

coordination costs and are not highly-available. In contrast, BASE (Basic Availability, Soft state, and Eventual consistency) transactions [12, 61, 87, 105, 128] are a class of transactions that can be implemented using asynchronous execution only. The same atomicity and durability aspects of ACID transactions are desirable, however, isolation properties are weakened to allow transactions to execute without coordination.

Many systems have taken the approach of combining transactions and causal consistency [12, 87, 130]. The idea is to have transactions reading from a causally-consistent snapshot, and executing the transaction on that snapshot, i.e. all effects of the transaction are ordered after the snapshot of the transaction. The updates are applied atomically in the local replica, and propagated asynchronously to remote replicas, where they are also applied atomically.

Eiger [87] is a successor of COPS that has limited support for transactions. The system supports read-only and write-only transactions, by employing a non-blocking variant of the two-phase commit protocol. The authors extended the model of COPS to track dependencies at the grain of transaction, instead of tracking dependencies per object. Eiger requires a coordinator for committing transactions. RAMP [12] overcomes this limitation by decoupling transactions propagation and visibility, allowing any client to detect isolation violations during execution, and complete ongoing dependent transactions before continuing.

SwiftCloud also allows the execution of transactions from a causal snapshot and committing them without using a coordinator. Transactions in SwiftCloud are applied immediately at the local replica and propagated to other replicas asynchronously. Any replica that receives a transaction can apply it locally without coordinating with other replicas, and can also forward the transaction to other replicas themselves, in case the origin replica fails.

Limitations of BASE transactions: Weakening the isolation properties of transactions might affect the correctness of applications. Typically BASE transactions never abort due to concurrency conflicts to ensure high availability, thus if a transaction commits locally, it will eventually commit at all replicas. Nonetheless, the effects of operations might be conflicting, in which case a convergence rule must be applied. This might lead to loss updates, or even if the system is able to integrate all concurrent updates, the semantic of the application might be broken.

For example, consider a relational database that has two tables $A(a,b)$ and $B(b,c)$, where a is primary key in table A and b is primary key in table B. The two tables are related by a referential integrity property that enforces that any value of column b of table A exists in table B. An operation $addToA(a,b)$ adds an entry to table A with values (a,b), an operation $remFromB(b)$ removes the element with identifier b. Under weak consistency, $addToA(a,b)$ and $remFromB(b)$ might execute concurrently, if the state of the database (in each replica) allows the execution of the operations, i.e. when any element containing b as primary key exists in B . If the two operations execute concurrently, merging the effects

of the two operations leads to an invalid database state because it adds a tuple (a, b) to A, assuming that b is a primary key of B, but that value has been removed concurrently.

3.3 Using sporadic coordination

Operations might have different consistency requirements depending on their semantics. Some operations might be able to execute safely under weak consistency, while other operations might require strong consistency to ensure that invariants are preserved. In this section we analyze the classes of application invariants that can be maintained using weak consistency, and what classes of invariants require stronger forms of consistency. Then, we discuss different approaches that combine strong and weak consistency to ensure application correctness without sacrificing availability. We discuss three different approaches: $Bloom^L$ is a programming language that provides constructions for writing coordination-free applications; Parallel Snapshot Isolation is a consistency model that provides strong consistency for operations that execute within a data center and propagate operations across sites in causal order; and finally, we discuss Red-Blue consistency, a system that allows the programmer to decide what is the most appropriate consistency model for executing different operations.

3.3.1 Invariant preservation under weak consistency

The availability of an application is conditioned by the ability of a system to enforce application correctness without using coordination to execute operations. This is a problem for application programmers as weak consistency guarantees are not well defined, allowing for inconsistencies that are difficult for programmers to correct.

Bailis proposed invariant confluence (*I-Confluence* [11]), as a property that determines whether an applications requires cross-replica coordination for correct execution. The property can be used to prove analytically if any execution allowed in an application under weak consistency is convergent and preserves the invariants, or it requires some form of strong consistency to maintain correctness.

The idea behind the approach is that ACID guarantees are not always necessary to ensure the correct execution of applications. The *I-Confluence* property can be used to find what invariants in an application hold on top of weak consistency.

The authors of *I-Confluence* checked what integrity constrains in SQL database systems are *I-Confluent*. The summary of their results is presented in Table 3.1. Most of the integrity constraints in SQL systems are eventually convergent, only a few operations are not compliant, as we discuss next.

Sequential identifiers and auto-increment fields cannot be implemented without coordination, as the outcome of operations depends on the order of execution. For example, if unique sequential identifiers are generated by replicas using a replicated counter, two concurrent operations might end assigning the same identifiers to different rows, which

Invariant	Operation	I-C
Attribute Equality	Any	Yes
Attribute Inequality	Any	Yes
Uniqueness	Choose specific value	No
Uniqueness size	Choose some value	Yes
AUTO INCREMENT	Insert	No
Foreign Key	Insert	Yes
Foreign Key	Delete	No
Foreign Key	Cascade Delete	Yes
Secondary Indexing	Cascade Delete	Yes
Materialized Views	Cascade Delete	Yes
$>$	Increment $[Counter]$	Yes
$<$	Increment $[Counter]$	No
$>$	Decrement $[Counter]$	No
$<$	Decrement $[Counter]$	Yes
$[Not]$	Any $[Set, List, Map]$	Yes
Size $=$	Mutation $[Set, List, Map]$	No

Table 3.1: *I-Confluence* analysis results for SQL integrity constraints. Transcript from Bailis et al. [11].

would violate uniqueness. Identifiers that do not need to be sequential can use some local information to disambiguate the values that are generated concurrently [103], for instance, by adding a per-host prefix. Foreign keys are *I-Confluent* in the general case, except for concurrent insertions and deletions involving the same constraint, as exemplified in section 3.2.4.3. Numerical invariants and other constraints that involve limiting the number of times certain operations may execute, like constraining the number of elements in aggregations, or collections, cannot be preserved under weak consistency, as exemplified in section 3.2.3.

I-Confluence provides a formal framework that can be used to write the proof that some invariant is preserved without coordination. To that end, it is necessary to specify the invariants of the application and determine the operations effects that would lead to invariant violations, which might be impractical for programmers. In our work, we are focused in automating the proof process, by automatically generating test cases that would lead to invariant violations. With our tool we still need to specify invariants and operations effects, but the tool automatically checks conflicts.

3.3.2 Supporting multiple consistency models

***Bloom^L*:** *Bloom^L [33]* is a logic programming language for programming eventually convergent applications. *Bloom^L* programs are sets of declarative statements about collections of facts. The language supports different types of statements that allow, for instance, storing persistent data or network communication. *Bloom instances* run the programs logic, provide a private storage and a communication interface, similar to server replicas. A particularity of the model is that instances only communicate using asynchronous

message passing.

To represent the facts in the database, the language provides support for lattice data-types that allow implementing, timestamps, sets, and maps, among others. The programmer may also specify his own lattices data types. These data-types, similarly to CRDTs, need to have a merge function that computes the least upper bound of two object instances, for convergence. Morphisms, monotone and non-monotone functions can also be defined for each lattice, for implementing general applications.

If programs only use morphisms and monotone functions, the application is convergent [7]. In contrast, the use of non-monotone functions in applications might lead to inconsistencies. The problem occurs because the evaluation of non-monotone functions results in non-determinism, allowing different instances of the application to diverge depending on the order in which messages are applied.

For verifying if applications are in fact convergent, $Bloom^L$ uses an analysis to check for non-monotone function calls. Upon finding a call to a non-monotone function, the algorithm signals that point in the code. These calls need to be modified to avoid using non-monotone functions, or the programmer must instrument that call with a coordination mechanism to execute that state-transition in coordination with other replicas. This analysis is based on the CALM theorem [7] which proves that logically monotone programs are convergent.

Parallel Snapshot Isolation: Snapshot isolation is a consistency model used in many production database systems [95, 102] that is weaker than serializability. In this model, transactions execute against the most-recent database snapshot, at the time the transaction starts. A transaction can commit if it has no write-write conflict with a concurrent transaction. Snapshot isolation allows write-skew anomalies [23]. This type of anomaly is characterized by two concurrent transactions have an intersecting read set, but write to two different objects. This anomaly is not considered a conflict under Snapshot isolation, however it might lead to invariant violations in applications (the referential integrity violation example, is an instance of this problem). A solution to circumvent this issue is to promote read operations to write operations [47] (e.g. by touching the read value), for transactions that the programmer know that might suffer from this anomaly.

Typically, snapshot isolation implementations require coordination across replicas for getting the database snapshot and for committing transactions, which amounts for multiple round trip times. To avoid contacting remote replicas, parallel snapshot isolation [117] is a relaxed form of snapshot isolation that allows executing transactions completely locally in some cases. Each object has a preferred site, which is the site that contains the most recent version of each object. When executing a transaction in some replica, the system generates a local snapshot of the database, which might contain stale versions of objects whose preferred sites are remote. If the transaction only modifies objects whose primary replica is the local replica, the transaction can commit locally,

because all possible write-write conflicts can be checked locally. In this case, committed transactions are replicated asynchronously to other replicas, respecting causal order. When a transaction modifies objects whose primaries are in remote replicas, the transaction uses a cross-replica commit protocol, similar to two-phase commit, to commit those changes in the preferred sites.

Red-Blue Consistency: In Red-Blue Consistency [83] the programmers chooses whether to use strong consistency or weak consistency for executing different operations. The insight for combining two consistency modes is that in many applications most operations can use weak consistency while only a few operations require strong consistency. Supporting different consistency levels allows using strong consistency only when necessary to maintain the application invariants, without affecting availability and latency in the general case.

The system distinguishes two types of operations, *red* and *blue* operations. *Blue* operations execute under causal consistency, while *red* operations must be ordered against each other (and after local *blue* operations). The implementation of *blue* operations is similar to the implementation of operation-based CRDTs. In the generation phase, a local replica executes the logic of the operation and computes its side-effects. The side effects are applied to storage in the shadow phase, which executes localy and remotely. *Blue* operation are propagated to all remote replicas in causal order, executing only the shadow phase to reproduce the effects of the original execution. *Blue* operations must be commutative by definition, otherwise they are flagged as *red*. *Red* operations execute across all replicas atomically to ensure that they are ordered across replicas.

Many operations can be implemented in a commutative fashion. However, in some cases, that could lead to invariant violations. In those cases, despite being commutative, programmers must flag those operations as *red*, to prevent those violations from occurring.

Deciding if operations are *red* or *blue* might be difficult for programmers, as it requires them to analyze the execution of concurrent operations manually. Sieve [84] automatizes the process of rewriting applications implemented on top of SQL databases to use *red-blue* consistency, deciding automatically if an operation is *red* or *blue*.

The approach uses a static analysis tool to determine the safety of operations with respect to the invariants of the application. To that end, Sieve analyses the blocks of code to determine the pattern of execution of each operation, to detect possible invariant violations. If there are any invariant that might be violated due to concurrent executions, the system marks those operations as *red*. For the remaining operations, Sieve makes them commutative.

3.3.2.1 Limitations

Bloom^L and *red-blue* consistency provide programming models that allow extracting more concurrency without breaking the correctness of applications. Solutions that require programmers to use a new programming model are hard to be adopted, due to the additional effort. Not only re-implementing applications is hard and error prone, but also programmers need to implement applications in a smart way to take the advantages of the approach. In the case of *Bloom*^{L}, the CALM analysis does not give any hint on how to implement applications correctly, forcing programmers to think of alternative implementations, which may not be obvious. Sieve can identify Red operations and generate Blue operations automatically, which is helpful for programmers.

Walter [117], a system that implements parallel snapshot isolation, is only capable of ensuring low-latency and consistency if operations only modify data in the same partition. On top of that, some invariants might not hold at all times, because the system only enforces causal consistency in the wide-area.

Separating operations that require weak and strong consistency allows to extract the benefits of both approaches, however, it still depends on coordination mechanisms to execute some operations.

We address the limitations of previous works by providing an analysis tool that is capable of identifying conflicting operations automatically, similarly to Sieve, and pursue two complementary alternatives for preventing invariant violations that try to avoid the use of coordination. In the first approach, we allow conflicting operations to execute without coordination in the general case, by moving the necessary coordination outside the critical execution path of operations. The idea is that replicas agree beforehand on which operations each replica can execute concurrently. In the second approach, we modify the logic of applications to make them *I-Confluent*, by using convergence rules to ensure that operations can be applied in any replica without violating invariants. In this case, we completely remove coordination from applications, but the classes of invariants that support this is smaller.

3.3.3 Reservations

The problem of invariant maintenance without coordination has been studied in the past in the context of mobile environments [72, 73]. The typical model for those type of systems comprises a set of (partial) replicas, the client devices, that work as surrogates of a primary replica that contains the full-state of the database. When connected to the network, a device can execute operations immediately at the central server. When disconnected, operations execute locally and are propagated to the server when the device becomes online.

The connectivity of mobile devices is very volatile, therefore, to execute operations, the server can only make few assumptions about the availability of clients. The consequence of having poor connectivity is that only very limited operations are able to execute

offline, to ensure that applications remain correct.

A common invariant in applications is to maintain quantities of resources over a certain limit, for instance to maintain the balance of bank accounts non-negative, or to avoid overselling items in a shop. In these applications, if a replicas does not have connectivity with the remaining replicas of the system, it can no longer process client requests to ensure invaraint preservation.

The escrow transactional model [93] is a model for executing transactions that allow replicas to operate offline and guarantees invariant preservation. The model was initially proposed for processing long-running transactions, but it was latter applied in the context of mobile computing to improve the semantics of applications that work partially offline [104, 115].

The idea is to grant permission to each replica to consume a portion of the available resources while guaranteeing that the sum of resources that can be consumed across replicas does not exceed the amount of resources available. To that end, a central server generates a number of consumable tokens and distributes them between replicas. Each token grants permission to a replica to execute some operation. For instance, each token may grant permission to a replica to sell one unit of an item. Tokens can be consumed offline, allowing replicas to process requests while disconnected from other replicas, as long as they have enough tokens. The downside of the approach is that, if some client has some tokens and disconnects forever, the tokens cannot be revoked by the server, since the client might consume them without informing the server. Also, when the server is unreachable, the basic implementation of the model, does not allow replicas to get tokens, even if they are able to contact each other. Implementations of the model have addressed this issue by using leases [104, 115], and supporting direct token transference. The idea of escrow objects can also be applied to other generic data-types that can partitioned [127], such as lists or stacks.

Figure 3.4 shows the typical architecture of a system that uses escrow transactions. A central server maintains information about resources and escrow distribution. A client has a local copy of the value and can use it locally as long as the consumed resources do not exceed the local escrow. When online, the client propagates the updates executed and may retrieve the remaining escrow for redistribution by the server.

Mobisnap [104] is a database system for mobile environments that generalizes the escrow model to allow sharing the access to fields in relational database systems. The system is composed by a central server that stores the database, and clients that run a limited version of the central server and host a partial replica of the storage.

Mobisnap provides a rich set of *reservations* specific for SQL databases:

- *Escrow reservations* are equivalent to those described before and can be applied to numeric data-types;
- *Slot reservations* grant exclusive permissions to some client to create some predefined record in the database, preventing conflicts with other records that might

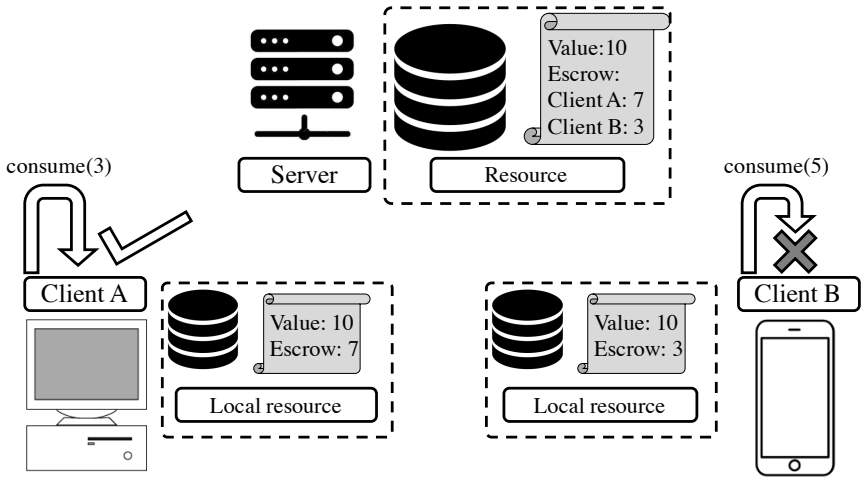


Figure 3.4: Escrow transactional model.

have overlapping values for some attribute;

- *Value-Change reservations* grant exclusive permissions to some client to modify some, or multiple, attributes in a record of the database;
- *Value-Use reservations* Gives permission to a client to use some attribute value regardless its concurrent modification.
- *Anti-Lock reservations* prevent any client from getting a reservation over some value.

Mobisnap introduces the use of leases to allow the system to automatically release reservations that are not used, renewed, or released by clients after some time. This is useful for preventing reservations from getting lost when clients go offline. Message delays can cause reservations to be released unexpectedly, even when the client is trying to commit some transaction that uses them. To mitigate that problem, if a client tries to commit some transaction without having all the necessary reservations, the central server will check if the necessary resources are available (i.e. non-reserved) to commit the transaction.

The implementation of Mobisnap relies on the central server for transmitting reservations between clients. This approach prevents clients from obtaining reservations to continue executing operations offline if the central server is unreachable.

Exo-leasing [115] is an implementation of the escrow model that allows clients to exchange tokens directly, without contacting a central server. In this approach, the logic for managing reservations is executed completely at the clients. Each token is also associated with a lease to allow releasing the tokens of clients that go offline for too long without releasing them, guaranteeing that they are not lost if clients fail.

Alternatively to the escrow model, the demarcation protocol [21] has been proposed to address the problem of managing resources in a distributed network. The main difference between the demarcation protocol and escrow transactions is that in the demarcation

protocol, the quantities are distributed across nodes, whereas in escrow, the total amount of available resources are known by the central server and clients only get permissions to consume them. In the demarcation protocol, the total available resources given by the sum of resources available in each server. For exchanging resources, replicas need to contact each other directly. This model can be used to enforce referential integrity constraints, and manage the generation of unique identifiers, by only allowing operations to execute operations if they have the necessary resources. We leverage the same principle to implement some of the reservation mechanisms that we propose in this thesis.

The reservation protocols for mobile environments do not address the replication of the server, imposing high latency for deployments in the wide area and no fault tolerance. We address these limitation in two ways: we replicate the storage system, and allow reservations to be distributed and managed by different replicas independently, providing low latency for clients that are close to some replica, and better availability by decentralizing reservations management and data storage.

3.4 Masking coordination costs

The solutions based on reservations prevent conflicting executions at the expense of providing worse latency when replicas do not have enough local reservations. In this section we discuss alternative solutions for executing transactions under strong consistency that hide the cost for committing those transactions. The idea is to extend the transactions execution model to allow clients to resume execution immediately after executing the code of a transaction, and commit the transaction in the background, at the expense of possibly of having to deal with transaction aborts at a later time.

3.4.1 Transaction chopping

The size of transactions is a factor that affects the performance of systems. Some transactions are small and execute very fast, with a small window of possible conflicts, other transactions are large, causing the whole system to slow down, either because they take locks for long periods of time [24], or because the transaction may have to re-execute multiple times before succeeding [76].

The idea of the SAGAS transactions model [51] is to divide a transaction into smaller transactions and commit each piece in sequence. Committing transactions in smaller pieces reduces the chance of conflicts for large transactions. If the commit of a piece aborts due to conflicts, or the logic of the application, the system might try to execute that piece of the transaction again, or rollback the effects of the transaction. Since a part of the effects of the transaction might have been exposed already, the system has to execute a *compensation* to revert the effects produced by the piece of the transaction. The downside of the approach is that the isolation of transactions is weakened because

concurrent transactions may observe the changes of committed pieces of a transaction before the whole transaction commits.

Helland et. al [62] and Brewer [26], independently, have acknowledged that compensations are an essential mechanism for implementing Internet services at a global scale with low-latency. They argue that it is impossible to prevent conflicts at a global scale without coordination, thus an alternative is to acknowledge that some things can go wrong, and provide support for repairing mistakes after the fact.

Transaction chains [131] uses transaction chopping to reduce the latency for executing transactions in geo-replicated systems. The objective of transaction chains is to minimize the observed latency of operations, while preserving serializability. In this system data is partitioned across multiple partitions. An operations is a sequences of small ACID transactions that only access data in a single partition, called a *transactions chain*. A static analysis determines which parts of a chain can be executed piecewise, while ensuring serializability. The analysis is based on the theory of transactions chopping [114], and consists in detecting the existence of dependencies between different pieces that need to be maintained. If two pieces of a transaction are not independent, coordination is required to ensure serializability.

The system only requires committing the first piece of the transaction before replying to the client, giving the "illusion" that operations execute fast. This can be achieved because subsequent pieces may abort only due to concurrency control, and can re-executed until they commit successfully.

3.4.2 Optimistic execution

In optimistic execution, applications reply immediately to users after finishing the execution of the transaction code and commit is handled in the background.

Bayou [122] allows speculative execution of transactions without coordination. Transactions commit tentatively at the local replica, and are eventually committed according to an unique order at all replicas. Users are allowed to access and modify a scratch replica while disconnected from the system, ensuring low latency for executing operations. Since replicas are able to update the state while disconnected, synchronization with remote replicas might lead to conflicts.

In Bayou, transaction conflicts are specified by the programmer at the logical level. For instance, the programmer can specify that a conflict occurs if two clients try to book the same room for a meeting overlapping in time. For each operation, the programmer provides the code of the transaction, a dependency-check and a merge procedure. When integrating the effects of an incoming operation, the dependency-check determines if the operation can be integrated in the local state, i.e., if it is not conflicting. If it can, then the operation is executed locally, otherwise the system executes the merge procedure to solve the conflict, for instance, by scheduling the meeting in an empty room.

Replicas execute transactions locally and their effects are propagate asynchronously

to other replicas in an anti-entropy process. Each replica maintains a stable log and a tentative log of operations that it has seen. Operations in the stable log do not change, while operations in the tentative log may have to re-execute in case operations that have to be ordered before that one arrive. A tentative operation becomes stable in some replica when it knows that there are no more tentative operation behind (for instance, by knowing the current state of other replicas).

Conceptually, the system might re-execute operations until all operations stabilize, but this is inefficient, because operations might have to execute many times if operations fall out of order, specially when replicas are offline for long periods of time. To speed-up the process, the system relies on a coordinator to establish the order of operations.

Bayou relies on a more complex programming model that requires programmers to specify conflict detection and conflict resolution code. Applications on top of Bayou appear to be responsive, because replicas execute operations locally, however the effects of operations might change in the future.

In PLANET [98] every transaction reply within predictable response times (after a timeout). A transaction runs through a number of phases and the programmer specifies the code to be executed if the transaction timeouts during one of those phases. If the transaction is not committed when the timeout is fired, it might still commit successfully in the background. The programmer can specify any behavior if the transaction has not yet completed. For instance, if the commit is still undergoing, the programmer can print the progress status of the transaction and wait for commit. The programmer may also choose to continue the execution of the application without waiting for the commit of the transaction. In that case, the system can send a notification to the application with the result of the transaction, and, in case the transaction aborts, the programmer can specify a piece of code to correct any possible mistakes.

To reduce the chance of aborting transactions at a latter time, the system determines the chance of successfully committing an ongoing transaction, based on statistical information from the underlying storage. This information is provided to the programmer, to help him decide decide whether the execution should continue speculatively or not, when a timeout occurs.

PLANET gives more flexibility for programmers to implement reliable applications in unpredictable environments, at the cost of a more complex programming model.

3.5 Discussion

In this work we build on classical techniques to enable geo-replicated systems to ensure high availability with stronger consistency guarantees than other systems that remain available under partitioning. Our approach consists in using static analysis to analyze which invariants can be maintained while avoiding coordination and modify applications accordingly. We propose two complementary approaches for modifying applications: the invariant violation avoidance approach, Indigo, which consists in applying specialized

reservations to prevent conflicting executions; and IPA, the invariant preserving approach, which consists in modifying operations to prevent invariant violations during replica reconciliation.

Figure 3.2 summarizes how our work compares to current state of the art solutions. We see that solutions that provide high availability and low latency cannot enforce invariants. IPA is able to do that by leveraging a static analysis, however the solution cannot be applied to all classes of invariants. Other solutions that use static analysis can provide a combination of high availability or low latency and invariant preservations, however they cannot enforce those properties simultaneously for all classes of operations. Indigo falls into this category, but further reduces the need for coordination, thus ensuring low latency, high availability and consistency for a larger number of operations. Below, we provide a more detailed comparison with existing solutions.

Weakly consistent data stores, such as Dynamo, Cops/Eiger and SwiftCloud provide high availability and low latency. Unfortunately, they can only provide support for a limited range of invariants [11] which makes them only usable in some applications. We propose the ***Bounded Counter*** [15] a data type that can be used to enforce numerical constraints on top of these systems.

Walter, $Bloom^L$ and Red-Blue/Sieve combine the benefits of strong and weak consistency. However, when operations may break invariants, these systems always have to pay the costs of coordination for every operation execution. $Bloom^L$ and Sieve use static analysis to flag operations as weak and strong automatically to make these approaches easier to use. Indigo follows the same principle, but uses static analysis to further reduce the number of cases where coordination is required, and reservation mechanisms to enhance the execution latency of operations in geo-replicated settings. None of these benefits are available in other systems that used reservations, like Mobisnap or Exo-leasing.

Bayou executes transactions speculatively in the local replica and establishes a total order of operations in the background. If concurrent operations conflict with each other, a conflict-resolution is applied to fix the conflict. In IPA, when possible, we modify the effects of operations to ensure that they never conflict. This solution has the benefit of avoiding re-executing operations multiple times.

Transaction chains and PLANET provide the illusion of executing operations with low latency to clients, while transactions synchronously commit in the background. In Transaction chains, the system ensures that transactions eventually commit by retrying to execute the individual pieces until committing every piece. In PLANET, clients do not need to wait for a transaction to commit before resuming execution. However, if the transaction eventually aborts, the client might have to execute some compensation. In IPA, we use compensation when modifying the effects of operations provides a bad semantics for the applications. The difference is that instead of applying effects preventively to avoids invariant violations, compensations correct the state of the database only when an invariant violation is detected.

Indigo and IPA are complementary approaches and can be combined together. For

Properties	Availability	Low latency	Invariant preservation	Static Analysis
Dynamo	High	Yes	No	No
Cops/Eiger	High	Yes	No	No
SwiftCloud	High	Yes	No	No
Bloom^L	Partial	Some	Yes	Yes
Red-Blue/Sieve	Partial	Some	Yes	Yes
Walter	No	Some	Yes	No
Mobisnap	Partial	Some	Yes	No
Exo-Leasing	Partial	Some	Yes	No
Tran. Chains	No	Yes*	Yes	Yes
PLANET	No	Yes*	Yes	No
Bayou	High	Yes*	Yes	No
Indigo	Partial	Some	Yes	Yes
IPA	High	Yes	Yes	Yes

Table 3.2: Comparison of state-of-the-art systems against Indigo and *IPA*.

instance, the semantics obtained by preventing certain conflicting executions might be more convenient in certain cases, but, in other cases, modifying operations can still provide a good semantics without impairing availability and latency in any case.

3.6 Final remarks

In this section we have described existing approaches that explore both sides of the availability/consistency trade-off. Solutions that privilege availability try to automatically ensure convergence without loosing availability. While it is possible to enforce substantial classes of invariants this way, some classes of operations are not compatible with the model. To solve that issue, some systems propose the use of coordination sporadically or mitigating the cost of coordination. In general these systems provide a more complex programming models, and cannot ensure low latency for executing all operations.

Solutions that use strong consistency always end up trading availability for classes of operations that are classified as conflicting. An important observation that we explore in our work is that not all instances of some operation will always leads to invariant violations. Previous works that require categorizing operations in conflicting or non-conflicting lead to excessive use of coordination in applications. In our work we show that is possible to execute some conflicting without coordination. The escrow transactional model and reservations already explored this insight, by allowing a limited number of operations to execute without coordination. We apply the model in the context of geo-replication, allowing to execute a wider range of operations this way.

We address the consistency/availability trade-off by combining static analysis of applications with specialized runtime mechanisms to ensure invariant preservation with low latency and high availability. We leverage static analysis to extract information from

applications automatically (with little effort for the programmer), and the specialized mechanisms allow the execution of operations efficiently. Our approaches are complementary and can be used together to provide rich semantics for applications while avoiding coordination in the general case.

We start the body of contributions of this thesis by presenting the *Bounded Counter*. The *Bounded Counter* is a data type that is capable of maintaining numeric invariants out-of-the-box, and provides a simple programming model. The solution can only be applied to single objects, which prevents it from being used to implement more complex applications. In the following chapters we extend the model to support transactions and discuss solutions that can be used to implement general applications.

Chapter 4

The Bounded Counter use-case

In geo-replicated systems, replicas can be distributed around the globe to reduce access latency for clients. To avoid paying the high costs for contacting replicas that are far apart for executing each operation, these systems typically use weaker consistency models that allow replicas to execute operations concurrently without coordination. A common approach for handling possible concurrency conflicts is to use *last write wins* registers, that only keeps a single value (according to some deterministic rule) for an object when it is written multiple times concurrently [78, 86, 126]. This semantics might be problematic for service providers, as it allows the loss of client updates, which might impact the functionality of the service. For instance, the system might lose an operation for buying some products if the product is bought multiple times concurrently. To address this problem, some commercial databases include reconciliation support for specific data types, such as counters in Cassandra and DynamoDB [78, 116], or CRDTs in Riak [121] that allow merging concurrent updates without losing them.

However, ensuring data type convergence is not a sufficient guarantee to enforce application-level correctness. This is an aspect that is many times overlooked by programmers who develop applications for weak consistency without understanding the limitations of the model [13]. Under weak consistency, concurrent operations might produce conflicting updates that, despite being convergent, might result in invalid states for the application. For instance, if an e-commerce application does not allow overselling products, i.e. the stock of some product must always be greater or equal to zero ($stock \geq 0$), simply checking that the stock is sufficient in the replica that receives an order request is not enough, as two concurrent operations executing in different replica might buy the last items, leading to a negative stock value. The difficulty of implementing applications that work correctly on top of weak consistency creates barriers to the adoption of the model by practitioners, which are forced to fallback to strong consistency models to implement

applications correctly.

In this chapter, we study the case of implementing applications with simple numeric constraints, like the one describe, on top of geo-replicated weak consistency systems. We propose a new data type design, the Bounded Counter, that support invariant preservation out-of-the-box. The *Bounded Counter* is capable of maintaining inequations in the form $x \geq K$, and can be implemented on top of existing replicated key-value stores depending on weak consistency only.

Our approach builds on the key idea of escrow transactions (see section 3.3.3), but is adapted to the context of geo-replication. Traditional implementations of the escrow model use a central server to keep track of all existing resources, and distribute them among clients. This model is not adequate for large-scale applications for two reason: first, it is difficult to partition resources over such a large number of clients, while trying to guarantee good distribution of the resources; second, the information about resources distribution and consumption might be lost in case of server, or client failures, due to the lack of replication.

In contrast to previous escrow implementations, ours includes no central authority, its totally asynchronous and supports replication. Instead of assigning escrows to the end-clients, in our approach, resources are assigned to each data-center, potentially in distinct geographical regions. The required information can be replicated internally to the data-center for durability. When a client contacts a replica for executing some operation, it consumes part of the escrow of the data center in which the operation executed. If the data center holds enough resources, the operation executes immediately at the local replica with low latency, otherwise it must coordinate with a replica in a remote data center to obtain more resources.

The management of replica's escrows is carried by a middleware deployed on top of the storage system. This middleware poses requirements in the underlying storage that are available in many existing systems. Our middleware implementation also serves as a cache that improves latency and write-throughput without reducing the fault tolerance properties of the underlying system.

To demonstrate the practicability of our approach, we have integrated our middleware with Riak, and evaluated the performance of the system in a geo-replicated environment. The evaluation shows that:

- When compared to strong consistency, our approach can enforce invariants without paying the latency price of replica coordination;
- When compared to unmodified weak consistency, we guarantee that invariants are not broken, and, with optimizations we were able to increase throughput with a small penalty in latency.
- Our middleware is capable of effectively managing replicas' escrow, providing low latency for operations in the general case.

The remainder of the chapter is organized as follows: Section 4.1 makes an overview of the approach; Section 4.2 explains how the *Bounded Counter* CRDT works; Section 4.3 presents the middleware that extends Riak with numeric invariant preservation; Section 4.4 evaluates our prototypes; Section 4.5 compares this work to other similar approaches; and Section 4.6 concludes the chapter.

4.1 System model

We target a typical geo-replicated scenario, with copies of application data and logic replicated in multiple data centers (DCs) scattered across the globe. End-clients contact the closest DC for executing operations. We consider that system processes are connected by an asynchronous network and assume that processes might fail by crashing. A crashed process either remains crashed forever, or recover with its persistent memory intact.

System API: In addition to *get(key)* and *put(key, value)* operations to access objects that are stored in the underlying database, our middleware provides the following operations to manipulate *Bounded Counter* objects:

- *create(key, type, bound)*, creates a new *Bounded Counter* with the given *key*, constraint *type* ($\geq, \leq$) and *bound*. E.g., *create('A', '≥', 10)* creates a counter with initial value 10 that enforces constraint $A \geq 10$;
- *value(key)*, returns the current value of counter *key*;
- *increment(key, value, remote)* and *decrement(key, value, remote)*, update the counter if it is known that the change will not break the invariant. The *remote* flag is used to block the operation until the replica receives resources from a remote replica or the information that no more resources are available. The operation return *success* if update succeeds or *error* otherwise.

Enforcing Numeric Invariants: To enforce numeric invariants, our design borrows ideas from the escrow transactional model [93]. The key idea is to see the difference between the value of a counter and its bound as a set of rights for executing operations. Consider, for example, a counter, n, with initial value $n = 40$ and an invariant $n \geq 10$, meaning that the value of n must be greater than 10 in any replica. In this case, there are 30 rights to execute decrement operations. Executing *decrement(5)* consumes 5 of these rights. Executing *increment(5)* adds 5 new rights. Rights can be split among the replicas of the counter, and exchanged between them. In our example, if there are 3 replicas, each replica can be assigned 10 rights. If the rights needed to execute some operation are sufficient in the local replica, the operation can safely execute without contacting other replicas, because it is guaranteed that the global invariant will not be broken. In the example, if the number of decrements executed in each replica is less or equal to

10, it follows that the total number of decrements will not exceed 30, and therefore the invariant is preserved. If a replica does not hold enough rights for executing a decrement, then, either the operation fails, or another replica must transfer some rights to increase the local replica's limit. In section 4.2.1 we explain how to do this using the *transfer* operation defined for the data type.

Our approach encompasses two components that work together to achieve the goal of our system: a novel data structure, the *Bounded Counter* CRDT, to maintain the necessary information for locally verifying whether it is safe to execute an operation or not; and a middleware that that proactively tries to ensure that the local limits of each replica always allow executing the requested operations.

Consistency Guarantees: We build our middleware on top of an eventually consistent database, extending the underlying guarantees with invariant preservation for counters. Our system guarantees that, despite replicas state be potentially divergent, the value of the counter never violates the bounds specified by the invariant, neither *locally* nor *globally*. By locally, we mean that the sum of all decrements of a replica r never exceeds the number of allowed operations (local increments + received rights): $\sum_i decrement_{r,i} \leq local\_rights_r$. By globally, we mean that, at any instant, the number of decrement and increment operations that are executed globally never exceed the limit: $init\_value + \sum_i increment_{r,i} - \sum_i decrement_{r,i} \geq K$. The local limits might change overtime, when replicas exchange rights between them, but the difference between increments and decrements can never never exceed K.

Note that the notion of causality is orthogonal to our design, in the sense that if the underlying storage system offers causal consistency, then we also provide numeric invariant-preserving causal consistency.

4.2 Designing the Bounded Counter CRDT

This section presents the *Bounded Counter*, a CRDT that maintains the necessary information for preventing the value of the counter to exceed some user-defined limit.

Conflict-free replicated data types (CRDTs) are a class of distributed data types that allow replicas to be modified without coordination, while guaranteeing that replicas converge to the same value after all updates are propagated and executed in all replicas. We explain in more detail the properties of CRDTs and different designs in section 3.2.2.

In this work, we adopted the state-based model of CRDTs, as we built our work on top of a key/value store (KV-Store) that synchronizes replicas by propagating the state of objects, instead of operations. In this model, an operation submitted in a given site executes in the local replica. The updated version of the object and the associated version information are propagated to the other replicas and merged with the state of those replicas.

Algorithm 6 *Bounded Counter* for invariant *greater or equal to K*.

```
1:  payload integer[n][n] R, integer[n] U, integer min
2:      initial [[0,0,...,0], ..., [0,0,...,0]], [0,0,...,0], K
3:  query value () : integer v
4:      v = min + Σ_{i∈Ids} R[i][i] − Σ_{i∈Ids} U[i]
5:  query localRights () : integer v
6:      id = repId()                                   ▷ Id of the local replica
7:      v = R[id][id] + Σ_{i≠id} R[i][id] − Σ_{i≠id} R[id][i] − U[id]
8:  update increment (integer n)
9:      id = repId()
10:     R[id][id] = R[id][id] + n
11: update decrement (integer n)
12:     pre-condition localRights() ≥ n
13:     id = repId()
14:     U[id] = U[id] + n
15: update transfer (integer n, replicaId to): boolean b
16:     pre-condition b = (localRights() ≥ n)
17:     from = repId()
18:     R[from][to] := R[from][to] + n
19: update merge (S)
20:     R[i][j] = max(R[i][j], S.R[i][j]), ∀i, j ∈ Ids
21:     U[i] = max(U[i], S.U[i]), ∀i ∈ Ids
```

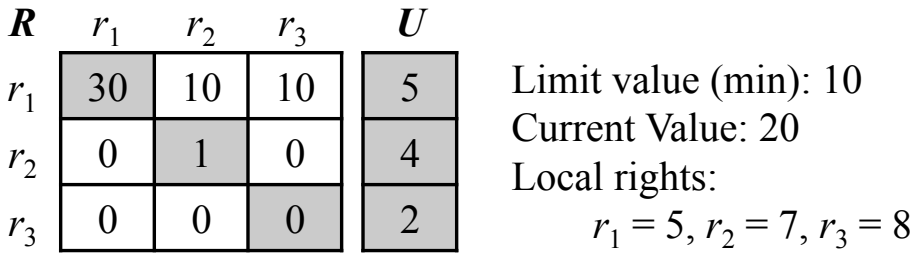


Figure 4.1: Example of the state of the *Bounded Counter* for maintaining the invariant *greater or equal to 10*. Increment operations are highlighted in matrix R, and decrements in U.

4.2.1 Bounded Counter CRDT specification

We now detail the design of the *Bounded Counter*, a CRDT for maintaining the invariant *greater or equal to K*. In section 4.2.3 we discuss how to extend the data-type to simultaneously support the invariant *less or equal to Q*. A pseudocode of the data type is presented in algorithm 6.

***Bounded Counter* payload:** The *Bounded Counter* maintains the limit value K and information about the rights each replica holds. For a system with n replicas, this information is stored in: a matrix R, where entry $R[i][j]$ records the rights transferred from replica i to replica j; and in a vector U, where $U[i]$ records the rights consumed by replica i.

Operations: An *increment* executed at r_i updates the number of increments executed in r_i by updating the value of $R[i][i]$. This operation is safe and can always execute locally.

A *decrement* executed at r_i updates the number of decrements executed in r_i by updating the value of $U[i]$. This operation can only execute if r_i holds enough rights locally before executing the operation, otherwise the operation fails.

The rights of replica r_i, returned by function *localRights*, are given by adding the local increments $R[i][i]$ to the transfers from other replicas to r_i, given by $\sum_{j:j\neq i} R[j][i]$, subtracting the transfers from r_i to other replicas, $\sum_{j:j\neq i} R[i][j]$, and subtracting the local decrements $U[i]$.

The operation *transfer* transfers rights from r_i to some other replica r_j, by increasing the value recorded in $R[i][j]$. This removes n rights for executing *decrement* from r_i and adds them to r_j. This operation can only execute if enough local right exist in r_i.

Figure 4.1 shows an example of a *Bounded Counter* for the invariant *greater or equal to 10*. The initial value of the counter is the bound of the constraint, 10. Replicas r_1, r_2 and r_3 have incremented the counter by 30, 1 and 0 units, respectively, as shown in the diagonal of R. The current value of the counter is given by adding to the limit, the increments performed in every replica, $\sum_i R[i][i]$, and subtracting the decrements, $\sum_i U[i]$, as represented in the grey cells. The example also shows that r_1 has transferred 10 rights to r_2 and r_3, which arerecorded in entries $R[1][2]$ and $R[1][3]$.

The *merge()* operation is executed during synchronization, when a replica receives the state of a remote replica. The local state is updated by just taking, for each entry, the maximum of the local and the received value.

4.2.2 Proof of correctness

For proving the correctness of *Bounded Counter*, it is necessary to show that all replicas of *Bounded Counter* eventually converge to the same state, i.e., that *Bounded Counter* is a correct CRDT, and that the execution of concurrent operations will not break the invariant.

A data type is a CRDT if it has the properties of a monotonic join-semilattice:

- The set S of possible states forms a join-semilattice ordered by $\leq$;
- The result of merging state s with remote state s' is the result of computing the least upper bound (LUB) of the two states in the join-semilattice of states, i.e., $merge(s, s') = s \sqcup s'$;
- The state is monotonically non-decreasing across updates, i.e., for any update u, $s \leq u(s)$.

As the elements of R and U are monotonically increasing (since operations never decrement the value of these variables), the join-semilattice properties are immediately satisfied – two states, s_0, s_1, are related by a partial order relation, $s_0 \leq s_1$, whenever

all values of R and U in s_1 are greater or equal to the corresponding values in s_0 (i.e., $\forall i, j, s_0.R[i][j] \leq s_1.R[i][j] \wedge s_0.U[i] \leq s_1.U[i]$). It is trivial that the merge function computes the LUB of the two states, since the function takes the maximum value of all entries in R and U, which are the smallest value possible of each entry. The maximum of each entry is idempotent, commutative and associative.

To guarantee that the invariant is not broken, it is only necessary to guarantee that a replica does not execute an operation (*decrement* or *transfer*) without holding enough rights to do it. Since operations execute sequentially and verify if the local replica holds enough rights before execution, it is necessary to prove that if a replica observes that it has N rights, it owns at least N rights. If replicas are well-behaved, the algorithms guarantees that line i of R and U is only updated by operations executed at replica r_i. Thus, replica r_i necessarily has the most recent value for line i of both R and U. As rights of replica r_i are consumed by *decrement* operations, recorded in $U[i]$, and transfer operations, recorded in $R[i][j]$, it follows immediately that replica r_i knows of all rights it has consumed. Thus, when computing the local rights, the value computed locally is always conservative (as replica r_i might not know about ongoing *transfers* to itself, from other replicas). This guarantees every decrement operation is safe and thus the invariant holds at all times.

We wrote the specification of *Bounded Counter* in TLA [80] and successfully verified that the invariant holds for all the cases that the tool generated.

4.2.3 Extensions

The exact same logic can be applied to preserve invariants of the form $x \leq K$: rights represent the possibility of executing *increment* operations instead of *decrement* operations, and the specification is changed accordingly to check that the number of increments does not exceed the defined limit.

Some applications may require two bounds for a counter, e.g., $Q \geq x \leq K$ A *Bounded Counter* can maintain an invariant of that form by combining the information of two *Bounded Counter*s in one object, similarly to what is done to specify a PN-Counter using two P-Counters [112].

In general, the approach used for *Bounded Counter*s can be adapted to other data types that support escrow [127].

4.3 Middleware for enforcing numeric invariants

We now present a middleware, depicted in Figure 4.2, that uses *Bounded Counter*s to extend cloud databases with numeric invariants.

In each DC, our system is composed by a set of nodes that compose the middleware layer and a a set of storage nodes that compose the key-value store. Operations on regular objects execute directly in the key-value store. Operations on counters are handled by

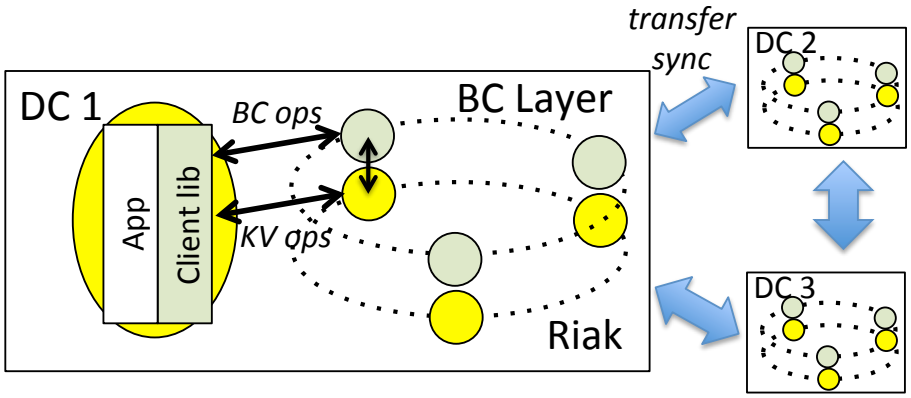


Figure 4.2: Middleware for deploying *Bounded Counters*.

middleware nodes, which in turn execute operations in the key-value store for persistence.

In our prototype, we use *riak_core* [74] to implement the middleware, and Riak 2.0, a key-value store inspired in Dynamo [40], as the underlying storage system. *Riak_core* provides a DHT communication substrate that is capable of executing logic in each node, while Riak 2.0 provides geo-replicated storage. Logical objects in Riak are replicated over a number of nodes inside each DC and across DCs. The rights for executing operations in the *Bounded Counter* are partitioned over the different DCs, for fault tolerance. However, under this strategy, concurrent updates inside the DC might affect the correctness of the *Bounded Counter*. For example, if during a reconfiguration of the key-value store, concurrent requests to the same counter are sent to two different nodes, replicas might be able to consume the escrow assigned to that DC concurrently. To avoid this problem, we use the conditional write mode of Riak 2.0, where a write from a client fails if there has been a concurrent write since the client's previous operation in the same DC. This mechanisms ensures serialization of updates, which guarantees that rights of a single DC are consumed in sequence, regardless of the replica that processes the operation.

The steps for executing an operation are the following: an operation in a counter is sent to the DHT node responsible for the counter; the DHT node executes the operation by reading the counter from Riak, executing the operation and writing back the new value, using the conditional write mechanism; The operation only succeeds if it is safe, i.e., if the local replica holds enough rights to guarantee the invariant is preserved and if the object was not changed since the last read.

Since Riak does not geo-replicate keys marked as strongly consistent, our middleware is also responsible for replicating *Bounded Counters* across DCs. To this end, each DHT node periodically propagates modified *Bounded Counters* to the remote DCs. When the payload is delivered on the remote DC, it is merged with the local state. This strategy batches a sequence of local operations on a single key and propagates them in a single update, saving bandwidth and processing.

Transferring Rights: Our middleware exchanges rights between replicas in two situations. First, when an operation cannot execute in a replica and the application has specified that remote replicas should be used. In this case, the DHT node executing the operation requests a transfer from a remote DC. To this end, it sends a message to a node in a remote data center, so that it executes a *transfer* operation in the *Bounded Counter*. Second, replicas proactively exchange rights in the background periodically to balance the rights assigned to each replica.

We tested different strategies for exchanging rights between data centers. The basic idea behind these strategies is to prevent the local resources of a replica to be exhausted, which would prevent the operation from executing immediately. In our implementation, when the permissions of a certain counter in a data center go below a certain threshold, it will request more resources from the replica that currently has more resources for the counter being analysed. The replica that receives the transfer request, gives a portion of the results, depending on the current value. We do not evaluate strategies for exchanging resources systematically. Strategies for sharing resources fairly remain as future work.

Fault tolerance: We now analyze how our middleware designs provide fault tolerance building on the fault tolerance properties of the underlying cloud database.

The cloud database is assumed to have sufficient internal redundancy to never lose its state in a DC. However, a failure in a node of the middleware layer may cause the DHT to reconfigure, with the possibly that two nodes temporarily accept requests for the same key. This does not affect correctness as we rely on conditional writes to guarantee that operations of each counter are serialized.

During a network partition, rights can be used in both sides of the partition – the only restriction is that it is impossible to transfer rights between any two nodes in different partitions. If an entire DC becomes unavailable, only the rights owned by the unreachable DC become temporarily unavailable. This contrasts with state-of-the-art strong consistency protocols [81], which can only serve requests if at least a majority of replicas (or a primary) is reachable. In our approach, any replica can serve requests if it owns enough rights or if it can gather the needed rights from reachable replicas.

Improving the performance of the middleware: Our prototype includes a number of optimizations to improve its efficiency. The first optimization is to cache *Bounded Counters* on the DHT nodes. This allows us to avoid reading the counter, when it is already in cache. Second, under high contention in a *Bounded Counter*, the design described so far is not very efficient, since an operation must complete before the next operation starts being processed. In particular, since processing an update requires writing the modified *Bounded Counter* back to the Riak database to ensure durability, each operation can take a few milliseconds to complete. To improve throughput, while the write to Riak is taking place, the requests received by the DHT node are processed using the cached counter. The system still writes the batched updates to storage before replying to the waiting clients,

but this strategy allows to execute a single write for multiple requests. Our evaluation shows that this strategy improves the throughput of the system by orders of magnitude.

4.4 Evaluation

We evaluated experimentally our prototype to address the following main questions. (i) How much overhead is introduced by our middleware? (ii) What is the throughput and latency for different levels of contention? (iii) What is the latency when the value is close to the invariant bounds?

4.4.1 Configurations and setup

In the experiments, we compare the following approaches:

***Bounded Counter* (BC):** The implementation of the *Bounded Counter* described in the chapter, that is capable of preserving numerical invariants and uses a middleware to manage the escrow of each data center.

Convergent Counters (Weak): This approach uses Riak 2.0 Enterprise Edition (EE), which supports native counters running under weak consistency and geo-replication. Native counters handle conflicts automatically inside the database layer, but do not support constraining the range of possible values.

Strongly Consistent Counters (Strong): This approach enforces invariant preservation by forwarding all operations from all clients to a single DC. The DC that acts as the primary uses the conditional write mechanism to serialize updates.

Our experiments comprised 3 Amazon EC2 DCs distributed across the globe. The average latency between DCs is presented in Table 4.1. In each DC, we use three m1.large machines with 7.5GB of memory for running the database servers and server-based middleware and run a sufficient number of m1.large machines for executing the clients, without reaching any bottleneck of the environment (1 per DC).

Data is fully geo-replicated in all DCs, with clients accessing the replicas in the local DC. Riak operations use a quorum of 3 replicas for writes and 1 replica for reads. In *Strong*, geo-replication is not used, data is stored in the US-East DC, which minimizes the latency for remote clients.

RTT (ms)	US-E	US-W	EU
US-East	-	80	96
US-West	83	-	163
EU	93	161	-

Table 4.1: Average RTT between Data Centers in Amazon EC2.

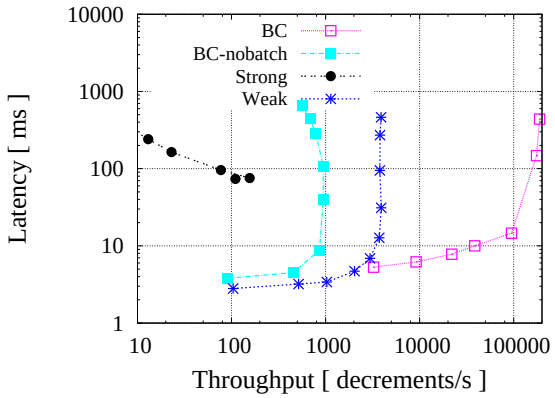


Figure 4.3: Throughput vs. latency for single counter.

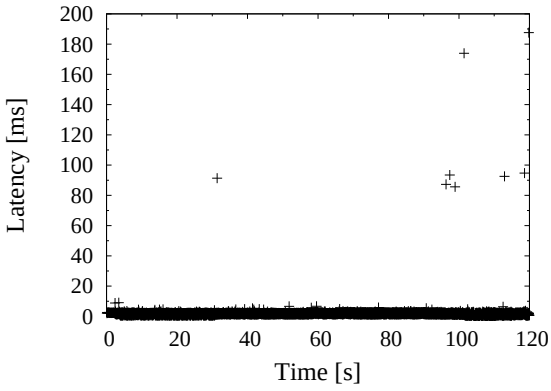


Figure 4.4: Latency of each operation over time for the *Bounded Counter* (BD).

4.4.2 Single counter

We first evaluate performance under high contention. To this end, we use a single counter initialized to a value that is large enough to never break the invariant. Clients execute 20% of increments and 80% of decrements in a closed loop with a think time of 100 ms. Each experiment runs for two minutes after the initialization of the counter. The load is controlled by tuning the number of clients running in each experiment, with clients evenly distributed among the client machines.

Throughput vs. latency: Figure 4.3 presents the variation of the throughput vs. latency values as more operations are injected in the system. The objective of this experiment is to evaluate the scalability of the different approaches.

The results of *Strong* show that throughput quickly starts degrading when load increases. This occurs because when more clients try to submit operations to a single DC they increase the interference, which prevents the conditional write from succeeding. We also observe that *Strong* exhibits the higher latency values which occurs because requests are all redirected to a single DC which is remote for 2/3 of the clients.

In comparison to *Strong*, the throughput of *Weak* is much larger and it does not degrade when increasing the load – after reaching the maximum throughput, increasing the load just leads to an increase in latency. The much higher throughput of the middleware solution is due to the batching mechanism of *BC*, which batches a sequence of updates into a single write to storage. To prove this hypothesis, we ran the same experiment, turning off the batching and writing every update in Riak, *BC-nobatch*. In this case, we can observe that the throughput is much lower than *Weak*, as the middleware introduces an additional communication step and executes operations in sequence. The same approach for batching multiple operations into a single Riak write could be used with other configurations, such as *Weak*, to improve their scalability.

Table 4.2: Latency of operations in each data center.

Median (Max) latency (ms)	Weak	Strong	BC
US-East	2 (7)	172 (180)	4 (9)
US-West	2 (7)	169 (187)	8 (13)
Europe	2 (8)	5 (9)	5 (11)

Latency under low load: Table 4.2 presents the median and maximum latency experienced by clients in different regions under low load. As expected, the results show that for *Strong*, remote clients experience high latency, while local clients are fast. It also shows that our middleware introduces an overhead of about about 2 ms when compared with *Weak*, which is justified by the additional communication steps.

Effects of exhausting rights: In this experiment we evaluate the behavior of our middleware when the value of the counter approaches the limit and contention for the last available rights rises. We initialize the counter with the value 6000 and 5 clients execute decrement operations until all rights are consumed. Figure 4.4 shows that most operations have low latency, with a few peaks of high latency whenever a replica needs to obtain additional rights. The number of peaks is small because most of the time the proactive mechanism for exchanging rights is able to provision a replica with enough rights before all local rights are consumed. We see these peaks more frequently near the end of the experiment, because there are less resources available and they might be temporarily exhausted. When all resources are consumed, replicas stop requesting rights and operations fail locally.

Invariant Preservation: To evaluate the severity of the risk of invariant violation, we computed how many decrements in excess were executed with success in the different solutions. We run the same experiment as before, but vary the number of clients. Figure 4.5 shows that *Weak* is the only configuration that experiences invariant violation. The operation for decrementing consists in reading the counter, checking if the value is greater than the limit and executing a decrement. The decrement operation is not atomic and because of this, multiple decrements can execute concurrently considering the same read value. This effect increases with the number of clients and concurrent updates.

4.4.3 Multiple counters

To evaluate how the system behaves in the common case where clients access to multiple counters, we ran the experiment of Section 4.4.2 with 100 counters. For each operation, a client selects the counter to update randomly with uniform distribution. The results presented in Figure 4.6 show that *Strong* now scales to a larger throughput. The reason for this is that by increasing the number of counters, the number of concurrent writes to the same key is lower, leading to a smaller number of failed operations. Additionally, when

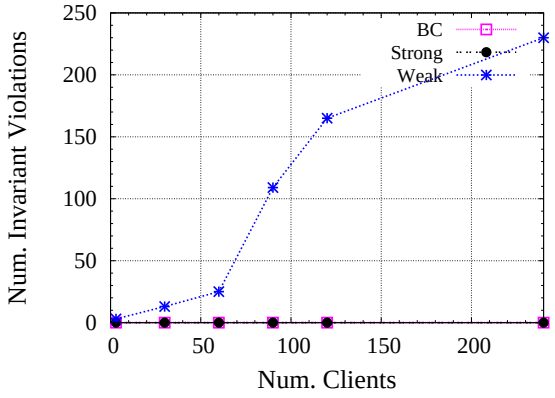


Figure 4.5: Number of decrements executed in excess.

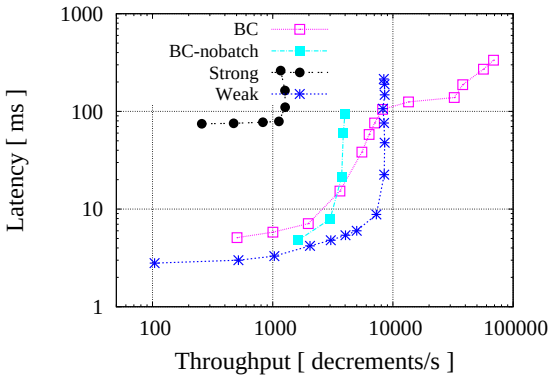


Figure 4.6: Throughput vs. latency with multiple counters.

the maximum throughput is reached, the latency degrades but the throughput remains almost constant.

The *Weak* configuration scales up to a much larger value (9K decrements/s compared with 3K decrements/s for a single counter). As each Riak node includes multiple virtual nodes, when using multiple counters the load is balanced among them – enabling multi-core capabilities to process multiple requests in parallel (whereas with a single node, a single virtual node is used, resulting in requests being processed sequentially).

The results show that *BC* has a low latency (close to that of *Weak*) as long as the number of writes can be handled by Riak's conditional write mode in a timely manner. In contrast with the experiment with a single counter, Riak's capacity is shared among all the keys, each contributing with writes to Riak. Therefore, as the load increases, writing batches to Riak will take longer to complete and contribute to accumulate latency sooner than in the single key case. Nevertheless, batching still allows multiple client requests to be processed per each Riak operation, leading to a better throughput. The maximum throughput even surpasses the results for the *Weak* configuration.

The results for *BC-nobatch*, where each individual update is written using one Riak operation, can be seen as the worst case of our middleware, in which the batching had no effect. Still, since all *BC* operations are local to a given DC and access only a quorum of Riak nodes, one can expect that increasing the local cluster's capacity should have a positive effect both on latency and throughput.

4.5 Related work

Many modern key-value stores offer support for rich data-types [22, 78]. Systems that are built on weak consistency, despite ensuring object convergence, are subject to application-level correctness anomalies, as the consistency properties that the models ensure are not not sufficient to ensure that applications behave correctly.

Our approach, contrarily to previous escrow implementations [104, 115], does not

require a central authority for managing resources. Replicas and middleware nodes, inside each data-center, may fail as long as the system has sufficient redundancy to be able to execute updates. In the case that network is partitioned and data-centers cannot contact with each other, the system can continue to execute local operations as long as there are enough local reservations.

Chrysanthis et. al have proposed other escrow data-types [127], including lists, and stacks. These data types can be incorporated in our middleware by implementing its interface.

Others have tried to reduce the need for coordination by bounding the degree of divergence among replicas. Epsilon-serializability [106] and TACT [129] use deterministic algorithms for bounding the amount of divergence observed by an application using different metrics: numerical error, order error and staleness. Consistency rationing [75] uses a statistical model to predict the evolution of replica state and allows applications to switch from weak to strong consistency upon the likelihood of invariant violation.

Holt et. al have recently proposed consistency types [66], which share the same principles of our approach. Consistency types can ensure the consistency level of the application in terms of application requirements. The authors implemented two policies. One ensures that operations returns within some target latency, and the other ensures that the error of a read operation is within some bounds. To implement the later approach, the authors proposed the design of a counter that enforces limit to the divergence of the real value, and a middleware similar to ours, that manages permissions to execute operations to guarantee error is within bounds. In comparison to our solution, consistency types cannot impose a strict limit for the value of object, but can ensure that the error does not exceed a certain limit, while our solution enforces a strict boundary.

4.6 Final remarks

Numerical invariants are commonly found in many Internet services. We can find instances of this type of invariant in e-commerce, flight reservation systems, advertisement and many other internet services. The *Bounded Counter* proposes a solution to ensure some forms of numerical invariants (inequations in the form $X \geq K$) under weak consistency, by forbidding the execution of operations that could lead to invariant violations. Our prototype imposes non-negligible overheads when compared to unmodified weak consistency systems, however, we expect that our approach performs better if implemented natively in the underlying systems, as the middleaware implementation introduces an additional communication step. Nonetheless, unmodified weak consistency systems are unable to maintain this type of invariant, and our approach performs much better than solutions that resort to strong consistency. Furthermore, optimizations introduced in the middleware layers improved the throughput of the system in orders of magnitude, by batching updates, with only a small latency penalty.

The solution that we presented is limited to the case that invariants do not span multiple objects. In the next chapter we will propose a general mechanisms for handling conflicting executions, which consists in detecting possible invariant violations through static analysis, and modifying the applications to never have conflicts during execution. Later, we leverage the *Bounded Counter* and a middleware, similar to the one implemented in this chapter, for achieving that goal.

Chapter 5

Explicit consistency

In this thesis we have made the case that it is difficult to ensure correctness of applications developed on top of weak consistency. The fundamental issue is that concurrent operations execute initially in a state that is isolated from the effects of other concurrent operations. Thus, when the effects of an operation are applied in other replicas, the conditions observed initially may no longer hold, which might result in an invalid database state. In this chapter, we present a principled theory to detect invariant violations that might occur due to concurrent executions. Our approach relies on the specification of application's operations and invariants to do be able to detect those violations through static analysis.

The underlying storage systems that support replicated applications are only capable of providing low-level consistency guarantees, such as restricting operation execution order, or ensuring system convergence, which are not sufficient to enforce application-level correctness, as seen in previous chapters.

Many works try to combine weak and strong consistency semantics to ensure application-level consistency, paying coordination costs when necessary for maintaining correctness [83, 117, 124], and using asynchronous operation execution otherwise. These works require programmers to classify operations in the two categories in order to ensure correctness and good performance, which is not a obvious choice in many cases. The classification separates operations that require coordination, from those that do not, which must execute under different coordination assumptions. However, as we have seen with the *Bounded Counter*, by leveraging information obtained during the execution, it is many times possible execute operations that are invariant-sensitive, while preserving correctness without coordination, thus, even for operations that require coordination, sometimes they should be able to execute without coordination, avoiding to pay the cost, when a local execution is safe.

We propose a new consistency model that departs from the difficult, and limited, approach of choosing weak or strong consistency for different operations. Our proposal only requires that the effects produced by operations maintain the invariants at all times, without making any assumptions on how the system executes those operations.

We introduce *explicit consistency* as an alternative consistency model, in which programmers identify the invariants of an application that the system must maintain at all times, and the underlying system uses the most appropriate mechanisms to execute operations to ensure those invariants.

Contrarily to traditional consistency models, that do not take into account the semantics of operations, and end up restring concurrency more than necessary for executing certain operations, in explicit consistency invariant-sensitive operations might execute concurrently in different replicas, while preserving correctness.

Through the analysis of application specifications, we are capable of pinpointing the operations that might lead to inconsistent states when executed concurrently. We do not impose the use of strong consistency for preventing those executions, instead, the programmer must modify the applications in any way that ensures correctness. In chapters 6 and 7, we show two different and complementary approaches that programmers can follow to achieve correctness while minimizing coordination. In the former, we generalize the use of the bounded counter (reservations), and in the latter we modify the code of operations to make them conflict-free in respect to concurrent operations.

We present a methodology for implementing explicit consistency in three steps: first, the programmer must specify application's invariants and the effects of operations; second, a static analysis checks the presence of conflicts in the application; finally, the programmer must modify the application in order to prevent the conflicts from occurring. Our methodology is backed by a tool that runs a static analysis to detect invariant violations automatically.

The organization of this chapter is the following: In section 5.1 we define explicit consistency; section 5.2 does an overview of the methodology and presents the running example; section 5.3 presents the language for expressing application correctness properties and the algorithm for detecting application conflicts; section 5.4 presents the proof of soundness of the algorithm; and, finally, we discuss implementation details of the algorithm in section 5.5.

5.1 Defining explicit consistency

In this section we present the system model and the definitions in which explicit consistency is built.

5.1.1 System model

We consider a database composed by a set of objects deployed in a typical cloud environment, where data is replicated in multiple data centers and partitioned inside each data center. We assume that each data center holds a full copy of the database. Clients interact with the system by requesting high-level operations that are executed in application servers running in each data center. We model the execution of an operation by distinguishing the execution phase, which only occurs at the origin, and the propagation phase, in which operation effects are propagated and applied to all replicas. More precisely, an operation is a piece of code that executes a sequence of reads and updates enclosed in a transaction. When the transaction first executes, its updates are recorded, and their effects are deferred until the transaction commits. Upon commit, the set of updates generated by the transaction are applied atomically on the local database state and also queued for replication. The propagation of updates between replicas is asynchronous and respects causal order. When a remote replica receives the set of updates of a transaction, it applies them atomically on the local database. Hereafter, when we use the term operation, it refers to the set of updates produced by the execution of the transaction code in the initial replica, and operation effects refer to the changes that are applied to all replicas.

We denote by $o(S)$ the state after applying the updates of operation o to some state S. We define a database snapshot, S_n, as the state of the database after executing a sequence of operations $o_1,\ldots,o_n$ from the initial database state, S_{init}, i.e., $S_n = o_n(\ldots(o_1(S_{init})))$. The set of operations reflected in a database snapshot S is denoted by $Ops(S)$, e.g., $Ops(S_n) = \{o_1,\ldots,o_n\}$. The state of a replica results from applying both local and remote operations, in the order received.

We say that an operation o_a happened-before operation o_b executed in state S, $o_a \prec o_b$, iff $o_a \in Ops(S)$, i.e. o_b executed in a state where o_a is visible. Two operations o_a and o_b are concurrent, $o_a \parallel o_b$, iff $o_a \not\prec o_b \wedge o_b \not\prec o_a$ [79].

For an execution of a given set of operations O, the happens-before relation defines a partial order among them, $\mathcal{O} = (O, \prec)$. We say $\mathcal{O}' = (O, <)$ is a valid serialization of $\mathcal{O} = (O, \prec)$ if $\mathcal{O}'$ is a linear extension of $\mathcal{O}$, i.e., $<$ is a total order compatible with $\prec$.

Operations can execute concurrently, with each replica executing operations according to a different valid serialization. This raises the problem that the state of the various replicas of the database could diverge, in case these operations do not commute. To prevent this, we assume the system gives the programmer the choice of various deterministic conflict resolution rules to achieve state convergence on a per-object basis, i.e., the result of applying updates that were executed concurrently is deterministic independently of the execution order. In our prototypes, we rely on conflict-free replicated data types (CRDTs) [112, 117] to achieve this goal.

We consider that application correctness can be expressed in terms of invariants. An invariant is a logical condition expressed over the database state. Some state S preserves

an invariant I iff $I(S) = true$, where $I(S)$ is a function that checks the validity of the invariant in state S. A state S_i is *I-valid* (or simply valid) iff $I(S_i) = true$. We say a state is *I-invalid* (or simply invalid) if it is not valid. We require that the initial state, S_{init}, is valid.

5.1.2 Explicit consistency definition

Explicit consistency is a novel consistency model for replicated systems, where programmers define the application-specific correctness rules that should be met at all times. These rules are expressed as invariants over the database state. For instance, the *Bounded Counter* invariant can be specified as a numerical inequality $x \geq K$.

Our formal definition starts with the helper definition of an invariant I, as a logical condition over the state of the database. We say that state S is an *I-valid state* if I holds in S, i.e., if $I(S) = true$.

Definition 5.1 (I-valid serialization)**.** *Given a set of operations O and its associated happens-before partial order $\prec$, $O_i = (O, <)$ is an* I-valid serialization *of $\mathcal{O} = (O, \prec)$* iff *O_i is a valid serialization of O, and I holds in every state that results from executing some prefix of O_i.*

We can now formally define the conditions that a system must uphold to ensure explicit consistency.

Definition 5.2 (Explicit consistency)**.** *A system provides explicit consistency* iff *all serializations of $\mathcal{O} = (O, \prec)$ are I-valid serializations, where O is the set of operations executed in the system and $\prec$ their associated partial order.*

This concept is related to the *I-Confluence* proposed by Bailis et al. [11]. *I-Confluence* defines the conditions under which operations may execute concurrently, while still ensuring that the system converges to an I-valid state. The current work generalizes this to cases where coordination is needed, and furthermore proposes efficient solutions to execute operations with low latency and high availability.

5.2 Overview

In this section we describe a methodology for implementing applications under explicit consistency. We briefly describe the two approaches that we explore in the following chapters, and present the running example that we will use throughout the following chapters to explain the technique.

5.2.1 Methodology

Given the application invariants, our approach for achieving explicit consistency is implemented in three steps:

i Detect the sets of operations that may lead to invariant violation when executed concurrently, called *I-offender sets*.

ii Select an efficient mechanism for handling I-offender sets.

iii Modify the application code to use the selected mechanism on top of a weakly consistent database system.

The first step consists of discovering I-offender sets. This step can be achieved through static analysis. We provide an analysis that analyses the specification of applications to detect I-offender sets. This information is provided by the application programmer, as annotations specifying the changes performed by each operation. Using this information, combined with the application invariants, the analysis infers the sets of operation invocations that, when executed concurrently, may lead to invariant violation. Conceptually, the analysis considers all reachable database states and, for each state, all sets of operation invocations that can execute in that state; it checks if executing these operations concurrently might cause an invariant violation. Obviously, it is not feasible to exhaustively consider all database states and operation effect sets; instead, a practical approach is to use efficient verification techniques. This is detailed in Section 5.3.

In the second step, the programmer decides which approach to use to handle the I-offender sets. We study two possible approaches: *violation avoidance* and *invariant preservation*.

Violation avoidance (Indigo, chapter 6): consists in restricting the execution of operations that could result in invariant violations. Given the information of conflicting operations, the programmer modifies the code of applications with mechanisms that prevent conflicting operations from executing concurrently at different replicas. We provide new reservation data-types to that end. The new reservations mechanisms can be used to prevent operations from executing concurrently based on the current state of the database and the parameters of operations. These mechanisms, despite constraining concurrency, allow more operations to execute concurrently than traditional approaches that enforce strong consistency. For instance, in the referential integrity example presented in section 3.2.4.3, operations $addToA(a, b)$ and $remFromB(b)$ are potentially conflicting. However if parameter b is different in each function call, these operations can execute concurrently. Our reservation mechanisms supports this behavior, without requiring cross-replica coordination, which would not be possible under strong consistency.

Invariant preservation (*IPA*, chapter 7): in this approach we do not constraint concurrency in any way. Instead, we modify the implementation of operations to guarantee that clients will never observe invalid database states. The insight of the approach is that in many cases invariant violations occur because programmers do not account for the effects of concurrent executions. We have observed that some of those violations can be

prevented by adding effect to operations and choosing appropriate convergence rules. To determine how the operations have to be changed, we provide an algorithm that automates the process of testing different variants of operations specifications to search for alternative specifications that are conflict-free. The modified operations have the same sequential semantics, but preclude invariant violations in the concurrent setting. For instance, in the example of referential integrity, the conflict occurs because the operation that adds the reference between table A and B assumes that the referred element in B will exist after the execution of the operation (because it existed in the local state, when the operation executed locally), however that is not guaranteed under weak consistency, as an operation in remote replicas might remove that element concurrently. To fix that violation, we can modify the $addToA(a,b)$ operation to recreate b in B and choose a converge rules for elements in table B that ensures that an $add(b)$ operation wins over a concurrent $remove(b)$ operation (add-wins).

In the third step, the application code is modified to use the right mechanisms to solve conflicts during runtime. In both approaches, applying the modifications to the code is as simple as adding few lines of code.

5.2.2 Running example

To explain how to implement explicit consistency we use the example of the Tournament management application.

The application allows the management of player, tournaments and the participation of players in tournaments. The application provides the usual *add* and *remove* primitives for tournaments and players. Players participate in tournaments by *enrolling* in them. Tournament phases transition between three possible states: *initialized*, *running*, or *finished*. When initialized players can enroll and *disenroll* freely from the tournament. After starting, players can compete against each other by playing *matches*. If a match is player between two players, they cannot disenroll from the tournament anymore. After finishing, no more players can enroll or disenroll in the tournament. Each tournament has a maximum number of players that has to be maintained at all times.

The specification of the application is presented in listing 5.1. In our prototype, we specify applications using Java annotations, to specify the effects of operations and the global invariants of the applications. We describe the language for specifying applications in section 5.3.1.

We chose to model our own benchmark application, instead of using reference benchmarks, such as TPC-C and TPC-W [37], because it allows us to define a wider range of invariants that are not available in these benchmarks.

```
1 @Unique("player(p)")
2 @Unique("tournament(t)")
3 @Inv("forall(Player:p, Tournament:t) :- enrolled(p,t) =>
4  player(p) and tournament(t)")
5 @Inv("forall(Player:p,q, Tournament:t) :- inMatch(p,q,t) =>
6  enrolled(p,t) and enrolled(q,t) and (active(t) or finished(t))")
7 @Inv("forall(T : t) :- nrPlayers(t) <= Capacity")
```

```
@Inv("forall(T : t) :- active(t)  => nrPlayers(t) >= 1")
@Inv("forall(Tournament:t) :- active(t) => tournament(t)")
@Inv("forall(Tournament:t) :- finished(t) => tournament(t)")
@Inv("forall(Tournament:t) :- not( active(t) and finished(t))")i
public interface TournamentApp {

  @True("player(p)")
  RESULT addPlayer(Player p);

  @True("tournament(t)")
  RESULT addTournament(Tournament t);

  @False("tournament(t)")
  RESULT remTournament(Tournament t);

  @True("enrolled(p, t)")
  @Increments("nrPlayers($1, 1)")
  RESULT enroll(Player p, Tournament t);

  @False("enrolled(p, t)")
  @Decrements("nrPlayers($1, 1)")
  RESULT disenroll(Player p, Tournament t);

  @True("active(t)")
  RESULT beginTournament(Tournament t);

  @True("finished(t)")
  @False("active(t)")
  RESULT finishTournament(Tournament t);

  @True("inMatch(p,q,t)")
  RESULT doMatch(Player p, Player q, Tournament t);
}
```

Listing 5.1: Specification of the tournament management system written in Java.

5.3 Conflict detection algorithm

In this section we present the algorithm for detecting conflicting operations (I-offender sets). We first introduce the language for specifying invariants and post-conditions, then we describe the types of invariants that can be represented in the language, and, finally, we present the pseudo-code of the algorithm and prove its correctness. We do not check application's code, or that the effects of applications are correctly specified. The programmer is responsible for ensuring that the application implement the specification correctly.

5.3.1 Defining invariants and post-conditions

Invariants Application invariant are described using first-order logic formulas. More formally, we assume the invariant is an universally quantified formula in prenex normal form[1]: $\forall x_1, \cdots, x_n, \varphi(x_1, \cdots, x_n)$.. First-order logic formulas can express a wide variety of constraints; we give some examples in Section 5.3.2.

The invariant uses general predicates, defined by the programmer, which represent some property of the database state. For instance, predicate $player(p)$ might represent the existence of a player p in the database, and $enrolled(p,t)$ that the player is enrolled in tournament t. Similarly, numeric restrictions can be expressed through the use of functions. For example, we may use $nrPlayers(t)$ (the number of players in tournament t) to limit the size of a tournament: $\forall t, nrPlayers(t) \leq 5$.

[1] Formula $\forall x, \varphi(x)$ is in prenex normal form if clause φ is quantifier-free. Every first-order logic formula has an equivalent prenex normal form.

Post-conditions We use post-conditions to express the effects of operations. An operation either produces all the effects that are specified or none. There are two types of effect clauses: predicate clauses, which state the new value of a predicate after the execution of some operation (stating whether the predicate is true or false after the execution of an operation); and function clauses, which define the relation between the initial and final values of a function after an operation executes. To give some examples, operation *removePlayer*(p), which removes player p from the system, has a post-condition with predicate clause $\neg player(p)$, stating that predicate *player* is false for player p. Operation *enroll*(p,t), which enrolls player p into tournament t, has a predicate clause, *enrolled*(p,t), set to true, and a function clause $nrPlayers'(t) = nrPlayers(t) + 1$ that states that the new value for $nrPlayers$ increases one unit.

The syntax for post-conditions is given by the grammar:

$$
\begin{array}{lcl}
post & ::= & clause_1 \wedge clause_2 \wedge \cdots \wedge clause_k \\
clause & ::= & pclause \mid fclause \\
pclause & ::= & p(o_1, o_2, \cdots, o_n) \mid \neg p(o_1, o_2, \cdots, o_n) \\
fclause & ::= & f(o_1, o_2, \cdots, o_n) = opr \mid opr \oplus opr \\
opr & ::= & n \mid f(o_1, o_2, \cdots, o_n) \\
\oplus & ::= & + \mid - \mid * \mid \ldots
\end{array}
$$

where p and f are predicates and functions respectively, over objects $o_1, o_2, \cdots, o_n$.

Although our grammar imposes that post-conditions are a conjunction of clauses, it is possible to deal with operations that have alternative side effects, by adding an operation with each alternative set of effects. For example, an operation φ with post-condition $\varphi_1 \vee \varphi_2$ could be replaced by operations op_1 and op_2 with post-conditions φ_1 and φ_2, respectively. Doing this separation does not affect the correctness of the application.

5.3.2 Expressiveness of application invariants

Our specification model can express significant classes of invariants, as discussed next.

5.3.2.1 Restrictions Over The State

An application can define the set of valid application states, using invariants that constraint the set of valid states. By combining user-defined predicates and functions, it is possible to address a wide range of application semantics.

Numeric constraints Numeric constraints may be used to set lower or upper bounds to object values. We have previously shown how to limit the number of enrolled players in a tournament by using a function that counts the number of enrolled players. Programmers can provide new functions that express any numeric inequality. For example, to ensure that a player does not overspend his (virtual) budget: $\forall p, player(p) \Rightarrow budget(p) \geq 0$, or to disallow an experienced player from participating in a beginner's tournament: $\forall t, p, enrolled(p,t) \wedge beginners(t) \Rightarrow score(p) \leq MAX\_BEGINNER\_EXP$.

Uniqueness, a common correctness property, may also be expressed using a counter function. For example, the formula $\forall p, \Rightarrow nrPlayerId(p) = 1$, states that p must have a unique player identifier. Whereas, the formula $\forall t, tournament(t) \Rightarrow nrLeaders(t) = 1$ states that a collection has exactly one leader.

Integrity constraints An integrity constraint specifies the relationships between different objects, such as the *foreign key constraint* in relational databases. In the tournament application there are various constraints of this type, for instance stating that enrollments must refer to existing players and tournaments, or that a match of some tournament requires the players to be enrolled in the tournament. If the tournament application had a score table for players, another integrity constraint might be that every table entry must belong to an existing player: $\forall p, hasScore(p) \Rightarrow player(p)$.

General constraints over the state An invariant may also capture general logic constraints. For example, consider an application to reserve meetings, where two meetings must not overlap in time. Using predicate $time(m, s, e)$ to state that meeting m starts at time S and ends at time e, we could write this invariant as follows: $\forall m_1, m_2, s_1, s_2, e_1, e_2, time(m_1, s_1, e_1) \wedge time(m_2, s_2, e_2) \wedge m_1 \neq m_2 \Rightarrow e_2 \leq s_1 \vee s_2 \geq e_1$.

5.3.2.2 Restrictions Over State Transitions

In addition to conditions over database states, our language also allows to represent restrictions over state transitions. Our approach to represent these constraints is to to turn them into an invariant over the state of the database, by materializing the different phases with distinct predicates[1, 97].

For example, we do this to represent the phases of a tournament: the start and finished phases are represented by two predicates, which cannot be true at the same time. The initialized state is represented by having both predicates set to false. A transition of state is represented by altering the value of those predicates. The application enforces that the state transitions are done in a particular sequence by checking the state of the database. Another example is to enforce that players may not disenroll from tournaments after playing a match against another player in the same tournament. For this, we add a constraint that enforces that the players must be enrolled in a tournament t if they played a matched against each other. The above rule can be specified as follows: $\forall p, q, t, inMatch(p, q, t) \Rightarrow enrolled(p, t) \wedge enrolled(q, t)$.

5.3.2.3 Existential quantifiers

Some properties require existential quantifiers, for instance to state that tournaments must have at least one player enrolled: $\forall t, tournament(t) \Rightarrow \exists p, enrolled(p, t)$. This can be easily handled, since the existential quantifier can be replaced by a function, using a

technique called skolemization. For this example, we may use function $nrPlayers(t)$ as such: $\forall t, tournament(t) \Rightarrow nrPlayers(t) \geq 1$.

5.3.2.4 Uninterpreted predicates and functions

The fact that predicates and functions are uninterpreted imposes limitations to the invariants that can be expressed. It implies, for example, that it is not possible to express reachability properties or other properties over recursive data structures. To encode invariants that require such properties, the programmer has to express predicates that encode coarser statements over the database, which lead to conservative concurrency. For example, instead of specifying some property over a branch of a tree, the programmer must define the property over the whole tree.

5.3.2.5 Specification example

Listing 5.1 shows how to express the invariants for the tournament application in our Java prototype. The invariants in the listing are a subset of the examples just discussed. Application invariants are entered as Java annotations to the application interface (or class), and operation side-effects as annotations to the corresponding methods. Our notation was defined to be simple to convert to the language of the Z3 theorem prover, used in our prototype.

5.3.3 Algorithm

To identify the sets of concurrent operations that may lead to invariant violations, we perform static analysis of operation's post-conditions against invariants. This analysis focuses on the case where operations execute concurrently from the same initial state. Although we assume that in a sequential execution the invariants hold[2], concurrent operation executions at different replicas may cause invariant violations

First, we check whether concurrent operations may result in opposite post-conditions (e.g., $predicate(x)$ and $\neg predicate(x)$), breaking the generic (implicit) invariant that a predicate cannot have two different values. For instance, consider operations $addTournament(t)$ with effect $tournament(t)$, vs. $remTournament(p)$ with effect $\neg tournament(t)$. These operations conflict, since executing them concurrently with the same parameter t leaves unclear whether player t exists or not in the database. The programmer may address this convergence violation by using a conflict resolution policy such as *add-wins* or *remove-wins*.

The remainder of the analysis consists in checking the effect of executing pairs of operations concurrently on the invariant. Our approach is based on Hoare logic [65], where the triple $\{I \wedge pre\}\ op\ \{I\}$ expresses that the execution of operation op, in a state where precondition pre holds, preserves invariant I. To determine if a set of operations

[2] This can be achieved by having a precondition such that an operation produces no side effects, if its sequential execution against a state that does not meet that precondition would violate invariants.

are safe, we substitute their effects on the invariant, obtaining I', and check that the formula I' is valid given that the preconditions to execute the operations hold.

For correctness, it is sufficient to detect invariant violations only for pairs of operations. The intuition why this is correct is that the static analysis considers all possible initial states before executing each concurrent pair, and therefore adding a third concurrent operation is equivalent to modifying the initial state of the two other operations.

To illustrate this process, we consider our tournament application, with the following invariant I:

$$I = \forall p, t, nrPlayers(t) \leq 5 \land enrolled(p,t) \Rightarrow player(p) \land tournament(t)$$

For simplicity of presentation, let us examine each of the conjuncts defined in invariant I separately. First, we consider the numeric restriction: $\forall T, nrPlayers(t) \leq 5$, to illustrate how to check if multiple instances of the same operation are self-conflicting. In this case, one of the operations we need to take into account is $enroll(p,t)$ whose outcome affects $nrPlayers(t)$. This operation has precondition $nrPlayers(t) \leq 4$, the weakest precondition that ensures the sequential execution does not break the invariant (see Footnote 2). To determine if this may break the invariant, we substitute the effects of running the *enroll* operation twice into invariant I. Then we check whether this results in a valid formula, when considering also the weakest precondition. In this example, this corresponds to the following derivation (where notation $I\langle f \rangle$ describes the application of effect f in invariant I):

$$\begin{aligned}
& I\ \langle nrPlayers(t) \leftarrow nrPlayers(t)+1 \rangle \\
& \langle nrPlayers(t) \leftarrow nrPlayers(t)+1 \rangle \\
nrPlayers(t) \leq 5\ & \langle nrPlayers(t) \leftarrow nrPlayers(t)+1 \rangle \\
& \langle nrPlayers(t) \leftarrow nrPlayers(t)+1 \rangle \\
nrPlayers(t)+1 \leq 5\ & \langle nrPlayers(t) \leftarrow nrPlayers(t)+1 \rangle \\
& nrPlayers(t)+1+1 \leq 5
\end{aligned}$$

The resulting assertion $I' = nrPlayers(t)+1+1 \leq 5$ is not ensured when both the initial invariant and the weakest precondition $nrPlayers(t) \leq 4$ hold. This shows that $enroll(p,t)$ is a self-conflicting operation, and it belongs to an I-offender set: $\{enroll\}$

The second clause of I is $\forall p, t, enrolled(p,t) \Rightarrow player(p) \land tournament(t)$. This case illustrates a conflict between different operations. In this case, we check whether concurrent $enroll(p,t)$ and $remTournament(t)$ may violate the invariant. Again, we substitute the effects of these operations into the invariant and check whether the resulting formula is valid, assuming that initially the invariant and the preconditions of the two operations

hold.

$$
\begin{array}{r}
I\ \langle enrolled(p,t) \leftarrow true\rangle\ \langle tournament(t) \leftarrow false\rangle \\
enrolled(p,t) \Rightarrow player(p) \wedge tournament(t)\ \ \langle enrolled(p,t) \leftarrow true\rangle \\
\langle tournament(t) \leftarrow false\rangle \\
true \Rightarrow player(p) \wedge tournament(t)\ \ \langle tournament(t) \leftarrow false\rangle \\
true \Rightarrow false \\
false
\end{array}
$$

As the resulting formula is not valid, another I-offender set is identified: $\{enroll, renTournament\}$.

We now present the complete logic to detect I-offender sets in Algorithm 7. This algorithm statically determines the pairs of operations that are conflicting, which can be detected as follows:

Definition 5.3 (Conflicting detection)**.** *Operations op_1, op_2, $\cdots$, op_n* conflict *with respect to invariant I if, assuming that I is initially true and the preconditions for op_1 and op_2 are initially true, the result of substituting the post-conditions of both operations into the invariant is not a valid formula.*

Algorithm 7 Algorithm for detecting unsafe operations.

```
Require: I : invariant; O : operations.
 1: C ← ∅                                          ▷ subsets of unsafe operations
 2: for op ∈ O do
 3:     if self-conflicting(I, {op}) then
 4:         C ← C ∪ {{op}}
 5: for op, op' ∈ O do
 6:     if opposing(I, {op, op'}) then
 7:         C ← C ∪ {{op, op'}}
 8: for op, op' ∈ O : {op, op'} ∉ C do
 9:     if conflict(I, {op, op'}) then
10:         C ← C ∪ {op, op'}
    return C
```

The core of the algorithm is made of auxiliary functions, which use the satisfiability modulo theory (SMT) solver Z3 [39] to verify the validity of the logical formulas used in Definition 5.3. Function *self-conflicting*$(I, \{op\})$ determines whether op is self-conflicting, i.e., if concurrent executions of op with the same or different arguments may break the invariant. Function *opposing*$(I, \{op, op'\})$ determines whether op and op' have opposing post-conditions. Function *conflict*$(I, \{op, op'\})$ determines whether the pair of operations break invariant I, by making it false under concurrent execution. These operations rely on the solver to check the validity of a set of formulas, namely the invariant, the preconditions, and the updated invariant after substituting the effects of both operations.

Algorithm 7 uses these functions for computing I-offender sets in three steps. The initial step (line 2) determines self-conflicting operations. The second step (line 5) determines opposing operations by detecting contradictory predicate assignments for any

pair of operations. The last step (line 8) determines other I-offender sets by checking if combining the effects of any two distinct operations raises an invariant violation. If it leads to a conflict, it adds the pair to the set of I-offender sets.

The number of test cases generated is polynomial in the number of operations, $\mathcal{O}(|O|^2)$. However, the satisfiability problem to be solved in each auxiliary function is, in the general case, NP-complete [68]. Z3 relies on heuristics to analyze formulas efficiently, in most cases. The results presented in Section 5.5 suggest that it is fast enough to be practical.

5.4 Proof of correctness

As explained before, our approach starts by identifying sets of operations that cannot be executed concurrently without coordination to ensure invariant maintenance. We assume that the concurrent execution of other operations will not violate invariants. We now prove that this is true. To this end, we are going to demonstrate that the concurrent execution of non-conflicting operations, i.e. the execution of operations that do not belong to any *I-offender* set, never lead the database to an invalid state. In other words, we check that the execution of any non-conflicting operation, concurrently with some *I-offender* operation is safe and maintains the invariant.

In the defined system model we assume that the local execution of an operation consists in applying its side effects, and that the resulting state is always *I-Valid*. For simplicity we consider that operations always produce side effects. This assumption can be implemented by precluding applications from executing an operation if its execution would result in an invalid state.

To set the basis for our proof, first we formally define what are non-conflicting operations under our model. As explained before, our tool verifies for each pair of operations if they are conflicting.

Definition 5.4. *A pair of operations is non-conflicting iff applying the effects of the operations, in any order, over an initial state, where the pre-conditions of each operation are valid, generates an* I-Valid *database state, i.e., for the set of operations of an application, OP, and the set of* I-Valid *database states S:*

$$\forall op_a, op_b \in OP, \forall s \in S : I(op_a(s)) = true \wedge I(op_b(s)) = true$$
$$\Rightarrow I(op_a(op_b(s))) = true \wedge I(op_b(op_a(s)) = true$$

By definition 5.4, any serialization of the effects generated by two concurrent non-conflicting operations produces a database state that is *I-Valid*.

We denote by OP_{nc} the set of operations that are non-conflicting, i.e, $\forall op_i, op_j \in OP$: $non\_conflicting(op_i, op_j) = true$. We denote by S_j the the set of states where operations op_j can execute without violating invariants. $\forall op_j \in OP_{nc} : S_j = \{s \in S : I(op_j(s)) = true\}$ and, finally, $S_{ij} = S_i \cap S_j$.

The algorithm presented in section 5.3.3 detects pairs of conflicting operations. The idea is to check if the serialization of those operations over an *I-Valid* state is still an *I-Valid* state. A graphical representation of the execution that is tested by the algorithm is given in figure 5.1.

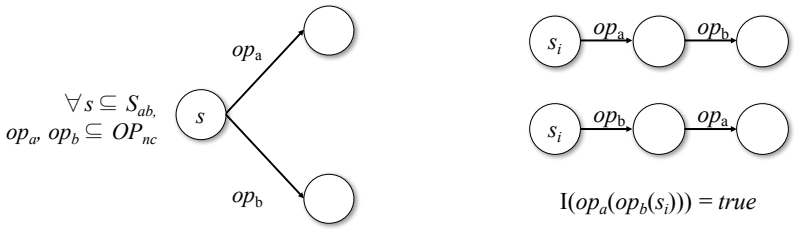


Figure 5.1: By definition 5.4, the execution of two non-conflicting operations on an *I-Valid* state can always be serialized into an *I-Valid* state.

Now, we are going to show, case-by-case, that it is always possible to generate a *I-Valid* serialization of any concurrent execution of any non-conflicting operations.

Figure 5.2 shows the case where one sequence of two non-conflicting operations executes concurrently with another operation. We will show that any serialization of these operations, in an order that respects the happens-before relationship, always leads to an *I-Valid* state.

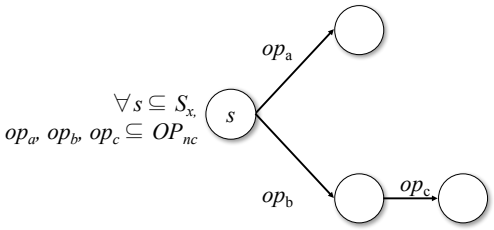


Figure 5.2: An operation op_a executes concurrently against operations op_b followed by operation op_c on an *I-Valid* state s.

Lemma 5.1. *For any non-conflicting operations $op_a, op_b, op_c \in OP_{nc}$, and any valid state $s \in S_x$, with S_x being the set of states where $I(op_a(s)) = true$ and $I(op_c(op_b(s))) = true$, any serialization of the operations op_a, op_b, op_c that respects the happens-before relation is an* I-Valid *serialization for initial state s, i.e.,*

$$\forall op_a, op_b, op_c \in OP_{nc}, \forall s \in S_x : I(op_a(s)) = true \wedge I(op_c(op_b(s))) = true \\ \Rightarrow I(op_a(op_c(op_b(s)))) = true$$

Proof. We know, from definition 5.4, that $\forall op_a, op_c \in OP_{nc}, \forall s_j \in S_{ac} : I(op_a(op_c(s_j))) = true$. If we show that $\forall s \in S_x : op_b(s) \in S_{ac}$, it follows that $\forall op_a, op_b, op_c \in OP_{nc}, \forall s \in S_x : I(op_a(op_c(op_b(s)))) = true$, which would prove lemma 5.1. To prove that $\forall s \in S_x : op_b(s) \in$

S_{ac} we must show that:

$$(i) : \forall s \in S_x : op_b(s) \in S_c$$
$$(ii) : \forall s \in S_x : op_b(s) \in S_a$$

If these conditions are true, it follows that $\forall s \in S_x : op_b(s) \in S_{ac}$, since $S_{ac} = S_a \cap S_c$.

From the execution, we know that (i) is true because op_c executed in state $op_b(s)$ for any $s \in S_x$.

As op_a and op_b are non-conflicting, $\forall s \in S_{ab} : I(op_a(op_b(s))) = true$. This implies that op_a can execute after $op_b(s)$, $\forall s \in S_{ab}$. Thus $op_b(s) \in S_a$, $\forall s \in S_{ab}$. We know that $S_x \subseteq S_{ab}$ because, as shown in figure 5.2, op_a and op_b can execute on any state $s \in S_x$. Since op_c executed after op_b, the set of states S_x must be contained in the set of states S_{ab}. This proves (ii). Clauses (i) and (ii) are true which proves lemma 5.1. □

Intuitively, what this proof shows is that if pairs (op_a, op_b), (op_a, op_c) are non-conflicting, and op_c executes after op_b for some execution, then op_a can execute after op_c to reach an *I-Valid* state. Given that concurrent operations are commutative, they can execute in any order that respects the happens-before relation, thus op_a can be serialized before or after op_b or op_c.

The next case that we are going to analyze consists in allowing the sequence of operations of lemma 5.1 to have an arbitrary length. This case is depicted in figure 5.3.

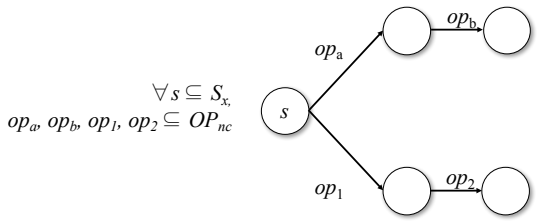


Figure 5.3: An operation op_a executes concurrently against a sequence of operations on an *I-Valid* state s.

For any non-conflicting operations $op_a, op_b, op_c \in OP_{nc}$, and any valid state $s \in S_x$, with S_x being the set of states where $I(op_a(s)) = true$ and $I(op_c(op_b(s))) = true$, any serialization of the operations op_a, op_b, op_c that respects the happens-before relation is an *I-Valid* serialization for initial state s, i.e.,

Lemma 5.2. *Given an initial state s in S_x, with S_x being the set of states where $I(op_a(s)) = true$ and $I(op_z(...(op_b(s)))) = true$, any possible serialization of those operations that respects the happens-before relation is an* I-Valid *serialization for initial state s, i.e.*

$$\forall op_a, op_b, ...op_z \in OP_{nc}, \forall s \in S_x : I(op_a(s)) = true \wedge I(op_z(...(op_b(s)))) = true$$
$$\Rightarrow I(op_a(op_z(...(op_b(s))))) = true$$

Proof. Following the same intuition behind the proof of lemma 5.1, we know that $\forall op_a$, $op_z \in OP_{nc}, \forall s_j \in S_{az} : I(op_a(op_z(s_j))) = true$. If we show that $\forall s \in S_x : op_{z-1}(...(op_b(s))) \in S_{az}$, it follows that $\forall op_a, ..., op_z \in OP_{nc}, \forall s \in S_x : I(op_a(op_z(...(op_b(s))) = true$, which would prove lemma 5.2. To prove that $\forall s \in S_x : op_{z-1}(...(op_b(s))) \in S_{az}$ we must show that:

$$(i) : \forall s \in S_x : op_{z-1}(...(op_b(s))) \in S_z$$
$$(ii) : \forall s \in S_x : op_{z-1}(...(op_b(s))) \in S_a$$

As in lemma 5.1, clause (i) is true because op_z executed in state $op_{z-1}(...(op_b(s)))$. We will demonstrate that clause (ii) is true by induction. The base case for induction is trivially true: $\forall s \in S_x, s \in S_a$ because op_a executed in state s. For the induction step, we need to show that the execution of a non-conflicting operations op_i in a state $s \in S_a \cap S_i$, S_{ai}, leads to a state belonging to $S_a \cap S_{(i+1)}$, $S_{a(i+1)}$, meaning that op_a and $op_{(i+1)}$ can execute after the execution of op_i, i.e.: $\forall op_i \in OP_{nc}, s \in S_{ai} : op_i(s) \in S_{a(i+1)}$. Given that op_a and op_i are non-conflicting, we have that $\forall s \in S_a : op_i(s) \in S_{ai}$, and, from the execution we have that $op_i(s) \in S_{(i+1)}$, therefore $\forall s \in S_{ai} : op_i(s) \in S_{a(i+1)}$. By induction (ii) is true. □

Intuitively, lemma 5.2 states that a concurrent operation can be serialized after any sequence of operations, given that those operations are non-conflicting.

Now, we analyze the case where each concurrent execution is composed by two operations in sequence. The execution is depicted in figure 5.4.

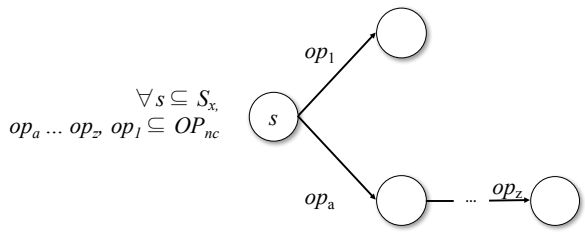


Figure 5.4: Two pairs of operations execute concurrently on an *I-Valid* state s.

Lemma 5.3. *Given an initial state s in S_x, with S_x being the set of states where $I(op_b(op_a(s))) = true$ and $I(op_2(op_1(s))) = true$, any any possible serialization of those operations that respects the happens-before relation is an* I-Valid *serialization for initial state s, i.e.*

$$\forall op_a, op_b, op_1, op_2 \in OP_{nc}, \forall s \in S_x : I(op_b(op_a(s))) = true \wedge I(op_2(op_1(s))) = true$$
$$\Rightarrow I(op_b(op_a(op_2(op_1(s))))) = true$$

Proof. We substitute the sequence of operations op_a followed by op_b by an operation op_{ab} that composes the effects of the two operations. This substitution is valid because these operations executed in sequence, and because we have shown in lemma 5.1 that a sequence of two non-conflicting operations can execute concurrently with a third operation without violating the invariant. From this lemma, it follows that the composed operation

and any other operation in a set of non-conflicting operations are non-conflicting. After applying the substitution to the formula, we have:

$$\forall op_{ab}, op_1, op_2 \in OP_{nc}, \forall s \in S_x : I(op_{ab}(s))) = true \wedge I(op_2(op_1(s))) = true \\ \Rightarrow I(op_{ab}(op_2(op_1(s)))) = true$$

which has been proved in lemma 5.1. □

The case of figure 5.5 depicts the situation where each concurrent execution is composed of a sequence of operations. This is the general case where each sequence of operations has an arbitrary size.

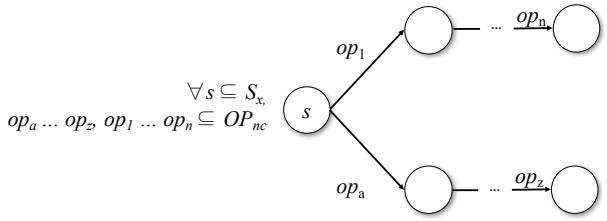


Figure 5.5: Two sequences of operations execute concurrently on an *I-Valid* state *s*.

Lemma 5.4. *Given an initial state s in S_x, with S_x being the set of states where $I(op_a(...(op_z(s)))) = true$ and $I(op_1(...(op_n(s)))) = true$, any possible serialization of those operations that respects the happens-before relation is an* I-Valid *serialization for initial state s, i.e.*

$$\forall op_a, ..., op_z, op_1, ..., op_n \in OP_{nc}, \forall s \in S_x : \\ I(op_z(...(op_a(s)))) = true \wedge I(op_n(...(op_1(s)))) = true \\ \Rightarrow I(op_z(...(op_a(op_n(...(op_1(s))))))) = true$$

Proof. The same intuition used for proving lemma 5.3 can be used to prove lemma 5.4. In this case, we rely on lemma 5.2 for substituting the sequence of operations by a single operation. □.

So far we have proved that any two concurrent executions of non-conflicting operations can always be serialized, leading to an *I-Valid* state. Showing that any number of concurrent executions of non-conflicting operations can be serialized to an *I-Valid* state ensures that our approach is correct, because it would guarantee that it is sufficient to prevent conflicting operations from executing concurrently to maintain application's invariants. The general case of execution is depicted in figure 5.6.

Theorem 5.1. *Given an initial state s in S_x, where S_x is the set of states where where $I(op_{an}(...(op_{a1}(s)))) = true \wedge ... \wedge I(op_{zn}(op_{z1}(s)))) = true$ any possible serialization of*

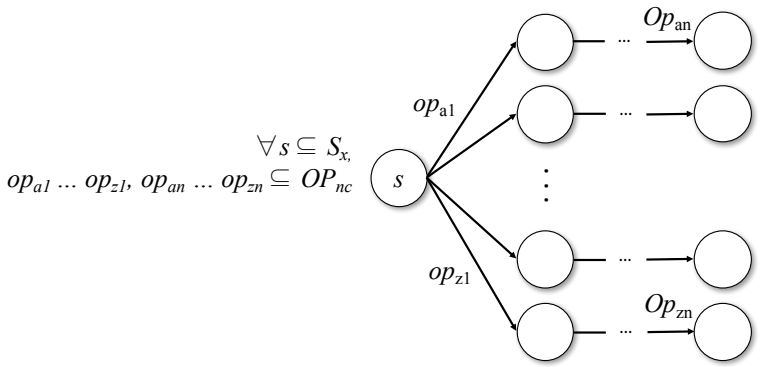


Figure 5.6: Any number of sequences of operations execute concurrently on an *I-Valid* state *s*.

those operations is an I-Valid *serialization for initial state s, i.e.*

$$\forall op_{1a}, ..., op_{nz} \in OP_{nc}, \forall s \in S_x :$$
$$I(op_{1z}(...(op_{1a}(s))) = true \wedge ... \wedge I(op_{na}(...(op_{nz}(s))) = true \Rightarrow$$
$$I(op_{nz}(...(op_{na}(op_{1z}(...(op_{1a}(s))))))) = true$$

Proof. Substituting any sequence of operations for a single operation, we just need to prove that:

$$\forall op_{1a}, ..., op_{nz} \in OP_{nc}, \forall s \in S_x :$$
$$I(op_{1az}(s)) = true \wedge ... \wedge I(op_{naz}(s)) = true \Rightarrow I(op_{naz}(op_{1az}(s))) = true$$

From lemma 5.4 we know that a sequence of non-conflicting operations can be executed after another sequence of concurrent operations. And this can be done for all sequences of operations, which recursively will lead to a *I-Valid* serialiazation of operations for the initial *I-Valid* state *s* □.

We have proven that non-conflicting operations can execute concurrently without generating invalid states, however some operations might be conflicting. *Reservations* allow to execute conflicting operations alongside non-conflicting operations without breaking invariants. The idea of reservations mechanisms is to control the execution of operations to ensure that only certain operations can execute concurrently, before those operations are visible to all participants. We can use reservations to control the execution of *I-offenders*. If we consider the set of operations that includes non-conflicting operations with a single operation from an *I-offender* set, the resulting set of operations is still non-conflicting (otherwise, the operation that would lead to a conflict would be in the *I-offender* set), thus it follows that the proposed approach maintains invariants when using reservations for restricting the execution of conflicting operations.

5.5 Tool and performance

We implemented the algorithm for detecting I-offender sets in Java, relying on the satisfiability modulo theory (SMT) solver Z3 [39] for verifying invariants. Our algorithm relies on the efficiency of Z3 to be able to analyze programs in reasonable time.

We implemented a prototype of the tool that provides an interface for programmers. The tool allows programmer to provide specifications, as the one showed in listing 5.1, and outputs conflicting pairs of operations, distinguishing the different types of conflicts. This information is helpful for guiding programmers.

Our prototype successfully found the I-offender sets in the tournament application. The average running time of this process in a recent MacBook Pro laptop was 730 ms for the more complex tournament application.

We have also modeled a few other applications that are used through this thesis. For instance, we have also modeled the TPC-W benchmark. This application has less invariants to check than our custom application, but has more operations. The running time for detecting I-offender sets was in this case 320 ms. These results show that although the average running time increases with the number of invariants and operations, our algorithm can process realistic applications in reasonable times.

5.6 Related Work

Weak consistency tends to be the preferred solution for building interactive global-scale services, as these provide lower latency and higher availability than strong consistency systems. However ensuring correctness under weak consistency is a challenging, as it requires programmers to reason about the possible outcomes of concurrent executions, which is a difficult and error prone task.

The static analysis of code is a standard technique used extensively for various purposes, including in a context similar to ours [30, 41, 71]. CALM analysis [7, 33], Sieve [84], and the homeostasis protocol [110] focus on analyzing applications to determine when coordination is required. Our approach is related to these, but does not enforce the use of coordination.

The CALM analysis [7, 33] pinpoints where exactly the system must use coordination to ensure convergence, by analyzing if operations are monotonic. This approach can only ensure that the state of a system is able to converge, but is not able to infer if the resulting state preserves the application invariants.

Sieve [84] uses analysis techniques to detect potential invariant violations, without requiring the programmer to specify application invariants. The analysis only addresses invariant violations arising from non-commutativity, which again does not preclude invariant violations. This work is complementary to ours, as the proposed techniques could be used to automatically infer application side effects.

The Homeostasis pursues a similar path, but its objective is to determine the maximum allowed divergence between replicas that does not affect the correctness of the system. The benefit of the approach is that is does not require the programmers to provide information about the invariants of the application, but the types of invariants that it can provide are more limited.

Similarly, *I-Confluence* [11] proposes a rule to determine what types of invariants can be preserved on top of weak consistency. This is the work that is most similar to ours, but they only provide an analytical model to prove what invariants can be maintained under weak consistency, which remains difficult for the general programmer to use. In contrast, our approach intends to automatize the process of detecting conflicts in applications, based on the specification of applications. While our work remains far from being an approach that programmers can use in production, it is an important step towards the verification of concurrent programs for distributed systems, and opens a number of interesting research questions.

5.7 Final remarks

In this chapter we proposed a consistency model, explicit consistency, that can help programmers to reason about concurrency conflicts with a principled approach. To support developing applications using this model, we proposed a methodology that can be used during the design phase of applications, and an auxiliary tool. Programmers specify the invariants of the application and the effects of each operation, and an algorithm statically infers what executions might possibly break the invariants of the application. With this information, programmers can take action to prevent those conflicts without necessarily resorting to a stronger consistency model.

In the following chapters we use explicit consistency to pursue two alternatives to ensure invariant preservation based on the information of conflicting operations. In the first approach, we prevent the execution of conflicting operations without requiring to explicitly check for conflicts across replicas, during execution. In the second approach, we propose modifying the effects operations to prevent them from generating conflicts when propagated to remote replicas. To that end, we rely on the conflict detection algorithm to check that the modified operations are conflict-free.

Chapter 6

Indigo

When operations execute under serializability the code of transactions is protected from external interferences that might affect the correctness of applications. Under this setting, the invariants of applications are maintained at all times given the code of the application is correct. This precludes many operations from executing concurrently, even though their effects would not break the correctness of the applications. However, it is difficult to decide which executions may lead to valid or invalid database states, forcing programmers to adopt conservative mechanisms that do not allow executing concurrent operations at different replicas.

The static analysis described in the previous chapter can help to exploit more concurrency from applications, while guaranteeing that applications are correct at all times. The analysis tells us what operations can effectively lead to invariant violations when executed concurrently, giving an opportunity to the programmer to avoid those executions when conflicts might arise, but allowing non-conflicting instances of those operations to proceed concurrently.

To provide efficient concurrency control, we augment the programming model of applications with mechanisms for preventing conflicting executions. Bringing concurrency control to an upper-layer allows to leverage the semantics of operations to control their execution. The disadvantage is that it also requires extra effort from programmers.

In this chapter we present Indigo, a system that uses reservations and CRDTs for implementing explicit consistency. Indigo handles I-offender sets in two ways: for simple opposing post-conditions we use CRDTs; for the remaining conflicting operations, we use reservation data types.

We provide new reservation mechanisms that are specially tailored for preventing invariant violations in geo-replicated settings. Similarly to lock mechanisms, reservations allow accessing protected data when a replica holds the necessary reservations. However

reservations can be shared among multiple replicas, allowing replicas to to access the same protected data concurrently when it is safe. When executing an operation, if a replica has all the necessary reservations, the operation executes immediately in the local replica ensuring low latency for that operation. If a replica does not hold enough reservations, it must contact remote replicas to acquire the necessary reservations. Furthermore, the transference of reservations can be done outside the critical path of execution, using asynchronous communication, to avoid contacting replicas during execution.

The modification of applications consists in using reservation mechanisms to handle I-offender sets. Reservations can be deployed at different granularity levels. For example, we can deploy reservations at the level of operation calls, or taking into accounts the parameters of operations to allow more concurrency. We present different designs of reservations that are suitable for maintaining different types of invariants, and explain the recipes for using them efficiently.

We have built Indigo, a middleware system that integrates the proposed reservation mechanism. Indigo can be implemented on top of existing key-value stores that provide transactions and ensure causality, allowing those systems to benefit from better consistency properties. The evaluation of our prototype shows that the mechanism can drastically reduce the latency of common operations, when compared to strong consistency. The catch is that the latency of some instances of uncommon operations might be penalized.

In section 6.1 we give more detail about the techniques used to avoid coordination; in section 6.2, we discuss the implementation of the reservation mechanisms; we proceed with the evaluation of the approach in section 6.3; and make our final remarks and discuss related work in section 6.5.

6.1 Handling I-offender sets

In the previous chapter we have seen how to identify I-offender sets — sets of operations that, when executed concurrently, might violate application invariants. These sets are reported to the programmer, who decides how each situation should be addressed. We now discuss the techniques that are available to programmers in Indigo.

6.1.1 Automatic conflict-resolution

One approach to fix I-offender sets is to allow the conflicting operations to execute concurrently, and automatically solve conflicts that might occur. Indigo has only limited support for this approach, since it can only address opposing postconditions conflicts. To this end, Indigo provides a library of objects that repair invariants automatically using crdt techniques proposed in the literature, e.g., sets, maps, graphs, trees with different conflict resolution policies [90, 112].

Application programmers may extend this library, in order to support additional invariants. For instance, the programmer might want to extend the unbounded set provided by the library, to implement a set with bounded capacity n. He could modify queries such that they ignore excess elements from the underlying unbounded set; however, he must take care to use a deterministic and monotonic algorithm to select the elements to ignore [89].

In the original Indigo, the programmer must ensure that the policies of the CRDTs do not generate any new inconsistencies in the database. In the next chapter, we discuss an extension to the conflict-detection algorithm that verifies if the CRDT policies used ensure that the database remains correct.

6.1.2 Invariant-Violation Avoidance

The other approach supported by Indigo is to avoid the effect of concurrent operations to violate invariants when they are applied together. Indigo provides a set of basic techniques for preventing this, which extend previous ideas from the literature [56, 93, 104, 115, 127]. In comparison to the previous work, we not only combine these ideas in the same system, but we also propose new implementations, which are optimized for a geo-replicated setting by requiring only peer-to-peer asynchronous communication, and relying on CRDTs to manage information [112].

6.1.2.1 Reservations

We now discuss the high-level semantics of the reservation techniques used to restrict the concurrent execution of operations. The next section discusses their implementation in weakly consistent stores.

UID generator: A very common invariant is uniqueness of identifiers [11, 83]. This problem can be easily solved, without coordination, by statically splitting the space of identifiers per replica. Indigo provides this service by appending a replica-specific suffix to a locally-unique identifier.

Multi-level lock reservation: The multi-level lock reservation (or simply multi-level lock) is our base mechanism to restrict the concurrent execution of operations that can break invariants. A multi-level lock is similar to a lock in the sense that it controls weather some operation, or piece of code, can be executed or not. However, it allows more concurrency than traditional locks [56]. Multi-level lock reservation have different types of rights, which give different guarantees about the actions that might occur in the system:

- *Shared forbid:* shared right to forbid some action to occur;
- *Shared allow:* shared right to allow some action to occur;
- *Exclusive allow:* exclusive right to execute some action.

If a replica wishes to execute an operation that is protected by a multi-level lock, first, it must acquire the type of right that is necessary to execute that operation. At any moment, different replicas might share the rights being used with other replicas, to allow other replicas to access the protected area. If a replica requires a different type of rights, for instance, to acquire an *exclusive allow*, first all replicas have to release their rights. The data-type ensures that at any time, the multi-level lock only has a single value globally.

We now show how to use this knowledge to control the execution of I-offender sets.

In the tournament example, $\{enrollTournament(P,T), remPlayer(P)\}$ is an I-offender set. To avoid the violation of invariants, we can associate a multi-level lock to each of the operations, for specific values of the parameters. For example, we can have a multi-level lock associated with $remPlayer(P)$, for each value of p. For executing $remPlayer(P)$, it is necessary to obtain the right *shared allow* on the reservation for $remPlayer(P)$. For executing $enrollTournament(P,T)$, it is necessary to obtain the *shared forbid* right on the reservation for $remPlayer(P)$. This guarantees that enrolling some player will not execute concurrently with deleting the same player. However, concurrent enrolls or concurrent removes are allowed. In particular, if all replicas hold the *shared forbid* right on removing players, the most frequent enroll operation can execute in any replica, without coordination with other replicas.

The *exclusive allow* right, in turn, is necessary when an operation conflicts with other operations and itself, i.e., when executing concurrently the same operation may lead to an invariant violation.

It would be possible to enforce any application invariants using only multi-level locks. However, in some cases it is possible to provide additional concurrency while enforcing invariants, by using the following types of reservations.

Multi-level mask reservation: For invariants of the form $P_1 \vee P_2 \vee \ldots \vee P_n$, the concurrent execution of any pair of operations that makes two different predicates false may lead to an invariant violation (if all other predicates were originally false). In our analysis, each of these pairs is an I-offender set.

Using simple multi-level locks for every pair of operations is too restrictive, as getting a *shared allow* on one operation would prevent the execution of all operations that could make any of the other predicates false. The reason why this is overly pessimistic is that, in this case, for executing an operation that makes some predicate false it suffices to guarantee that some other predicate remains true, which can be done by only forbidding the execution of operations that make it false.

To support this, Indigo includes a multi-level mask reservation that can be seen as a vector of multi-level locks. For the invariant $P_1 \vee P_2 \vee \ldots \vee P_n$, a multi-level mask with n entries is created, with entry i used to control operations that may make P_i false.

When a replica obtains a *shared allow* right in one entry, it must obtain a *shared forbid* right in some other entry. For example, an operation that may make P_i false needs to obtain the *shared allow* right on the i^{th} entry and a *shared forbid* right on an entry j for which the predicate is true. During execution, to find an entry to forbid, it is only

necessary to evaluate the current value of the predicate associated with each entry that can be locked.

Escrow reservation: We provide support for numeric invariants of the form $x \geq k$, with the *Bounded Counter*, with the same semantic that was described in chapter 4. This reservation can also be used for invariants of the form $x + y + \ldots + z \geq k$, where a single escrow reservation is sufficient to represent the expression. If one of the variables of the expression is involved in more than one invariant, the replica must also request reservations for those variables.

The variant called *escrow reservation for conditions* checks a count of elements against some condition; for instance, the number of participants in a tournament in the invariant $nrPlayers(T) < k$. In this case, if the same user is enrolled twice concurrently, two rights are consumed, although the number of participants increases by only one. This is conservative, but "leaks" rights. However, if the same user is disenrolled twice concurrently, then the number of users increases by only one; creating two rights might later let the invariant be violated.

Our escrow reservation for conditions addresses this problem using the following approach (considering invariant $c \geq k$). A decrement operation requires rights, just as a normal escrow reservation. However, an increment operation does not create rights immediately, but instead tags the reservation to be reevaluated. One of the replicas, marked as the primary for the reservation, is entrusted with recreating rights. To do so, it evaluates the distance between the current state and the threshold, taking into account the aggregate number of outstanding rights. More precisely, given the current value for $c = c_1$ and the number k_1 of outstanding rights (i.e., rights assigned to a replica and still not used, as known by the primary replica), $c_1 - k - k_1$ rights are created and assigned initially to the primary replica. This can be done either when the reservation is marked for reevaluation, or when new rights are needed.

Partition lock reservation: For some invariants, it is desirable to have the ability to reserve part of a partitionable resource. For example, consider the invariant that forbids two tournaments to overlap in time. Two operations that schedule different tournaments will break the invariant if the time periods overlap. Using a multi-level lock, it would be necessary to obtain an *exclusive allow* for executing any operation to schedule a new tournament.

However, no invariant violation arises if the time periods of concurrent operations do not overlap. To address this case, we provide a partition lock that allows a replica to obtain an *exclusive lock* on an interval of real values.[1] Replicas can obtain locks on multiple intervals, given that no two intervals reserved by different replicas overlap.

In our example, time would be mapped to a real number. To execute the operation that schedules a tournament, a replica would have to obtain a lock on an interval that includes the time from the start to the end of the tournament.

[1] Partition locks are a simplified version of partitionable objects [127] and slot reservations [104].

Invariant type	Formula (example)	Reservation
Numeric	$x < K$	Escrow(x)
Referential	$p(x) \Rightarrow q(x)$	Multi-level lock
Disjunction	$p_1 \vee \ldots \vee p_n$	Multi-level mask
Overlapping	$t(s_1, e_1) \wedge t(s_2, e_2) \Rightarrow s_1 \geq e_2 \vee e_1 \leq s_2$	Partition lock
Default	—	Multi-level lock

Table 6.1: Default mapping from invariants to reservations.

6.1.2.2 Using Reservations

A programmer, electing to use the coordination avoidance approach, must select the type of reservation to be used to avoid invariant violations. Figure 6.1 presents a default mapping between types of invariants and the corresponding reservations. Conservatively, it is always possible to resort to multi-level locks to enforce any invariant, at the expense of admissible concurrency, as discussed earlier.

When using multi-level locks to prevent the concurrent execution of I-offender sets, it is possible to use different sets of reservations. We call this a reservation system. For example, consider our tournament application with the following two I-offender sets, which follow from the integrity constraint associated with enrollment: $\{enrollTournament(P,T), remPlayer(P)\}$ and $\{enrollTournament(P,T), remTournament(P)\}$.

Given these I-offender sets, two alternative reservation systems can be used. The first system includes a single multi-level lock associated with $enroll(P,T)$, where this operation would have to obtain a *shared allow* right to execute, while both $remPlayer(P)$ and $remTournament(T)$ would have to obtain the *shared forbid* right to execute. The second system includes two multi-level locks associated with $remPlayer(P)$ and $remTournament(T)$, where enroll would have to obtain the *shared forbid* right in both locks to execute.

A simple optimization process is used to decide which reservations to use. As generating all possible combinations of reservation types may take too long, this process starts by generating a small number of systems using the following heuristic algorithm: (i) select a random I-offender set; (ii) decide the reservation to control the concurrent execution of operations in the set, and associate the reservation with the operation: if a reservation already exists for some of the operations, use the same reservation; otherwise, generate a new reservation from the type previously selected by the user; (iii) select the remaining I-offender set, if any, that has the most operations controlled by existing reservations, and repeat the previous step.

For each generated combination of reservations, Indigo computes the expected frequency of reservation operations needed, using as input the expected frequency of operations. The optimization process tries to minimize the expected frequency of reservation operations.

After deciding which reservation system will be used, each operation is extended to acquire the appropriate rights before executing its code, and to release appropriate rights

afterwards. For escrow locks, an operation that consumes rights will acquire rights before its execution (and these rights will not be released when the operation ends). Conversely, an operation that creates rights will create these rights after its execution. For multi-level masks, the programmer must provide the code that verifies the values of the predicate associated with each element of the disjunction.

6.2 Implementation

In this section, we discuss the implementation of Indigo as a middleware running on top of a causally-consistent store. We explain the implementation of the reservations mechanisms. First, we detail the protocol that the data-types implement and how it ensures correctness guarantees and how can they be used to enforce explicit consistency. Then, we discuss the implementation of the middleware, explaining how Indigo is designed to use an existing store.

6.2.1 Reservations

Indigo maintains information about reservations as objects stored in the underlying storage system. For each type of reservation, a specific object class exists. Each reservation instance maintains information about the rights assigned to each of the replicas; in Indigo, each datacenter is considered a single replica, as explained later.

The escrow lock object maintains the rights currently assigned to each replica, and the following operations modify its state: *escrow_consume* depletes rights assigned to the local replica; *escrow_generate* generates new rights assigned to the local replica; and *escrow_transfer* transfers rights from the local replica to some given replica. For example, for an invariant $x \geq K$, *escrow_consume* must be used by an operation that decrements x and *escrow_generate* by operations that increment x. For the escrow lock for conditions variant, a replica is tagged as the primary. The *escrow_generate* only creates rights in the primary.

When *escrow_consume* and *escrow_transfer* operations execute in a replica, if that replica has insufficient rights, the operation fails and it has no side effects. Otherwise, the state of the replica is updated accordingly and the side effects are asynchronously propagated to the other replicas, using the replication mechanisms of the underlying storage system. As operations only deplete rights of the replica where they are submitted, it is guaranteed that every replica has a conservative view of the rights assigned to it: all operations that have consumed rights are known, but operations that transferred new rights from some other replica may not be known immediately. Given that the execution of operations is serialized by the replica, this approach guarantees the correctness of the system in the presence of any number of concurrent updates in different replicas and asynchronous replication, as no replica will ever consume more rights than those assigned to it.

The multi-level lock object maintains which right (exclusive allow, shared allow, shared forbid) is assigned to each replica, if any. Rights are obtained for executing operations with some given parameters. For instance, in the tournament example, for removing player p the replica needs a *shared allow* right for player p. Thus, a multi-level lock object manages the rights for the different parameters independently. Each replica can hold a given right for a specific value of the parameters or a subset of the parameters values. For simplicity, in our description, we assume that a single parameter exists.

The following operations can be submitted to modify the state of the multi-level lock object: *mll_giveRight* gives a right to some other replica; a replica with a shared right can give the same right to some other replica; a replica that is the only one with some right can change the right type and give it to itself or to some other replica; *mll_freeRight* revokes a right assigned to the local replica. As a replica can receive rights by multiple concurrent *mll_giveRight* operations executed in different replicas, *mll_freeRight* internally encodes which *mll_giveRight* operations are being revoked. This is necessary to guarantee that all replicas converge to the same state.

As with escrow lock objects, each replica has a conservative view of the rights assigned to it, since all operations that revoke local rights are always executed initially in the replica that holds them. Additionally, due to causality, if ta replica is the only replica with some right, that information is true system-wide. This condition holds despite concurrent operations and the asynchronous propagation of updates, as any *mll_giveRight* executed in some replica is always propagated before a *mll_freeRight* in that replica.

The multi-level mask object is implemented using a vector of multi-level lock objects, with each operation specifying which multi-level lock must be modified.

The partition lock object maintains which replica owns each interval. When a lock is created, a single replica holds the complete interval of values. A single operation modifies the state of the object: *pol_giveRight*, which transfers part of the interval owned by the local replica to some other replica. Using the same reasoning as in the previous cases, it is clear that the local replica always has a conservative view of the intervals it owns.

6.2.2 Indigo Middleware

We have built a prototype of Indigo on top of a geo-replicated data store with the following properties:

- Causal consistency;
- Support for transactions that access a database snapshot and merge concurrent updates using CRDTs [112];
- Linearizable execution of operations for each object in each datacenter.

There are at least two systems that support all these functionalities: SwiftCloud [130] and Walter [117]. Given that SwiftCloud has a more extensive support for CRDTs, which

are fundamental for automatically repairing conflicts, we decided to build the Indigo prototype on top of SwiftCloud.

Storing reservations Reservation objects are stored in the underlying storage system and they are replicated in all datacenters. Reservation rights are assigned to datacenters individually, which keeps meta-data information small. As discussed in the previous section, the execution of operations in reservation objects at a given datacenter must be linearizable (to guarantee that two concurrent transactions do not consume the same rights).

The execution of an operation in the replica where it is submitted has three phases:

- The reservation rights needed for executing the operation are obtained; if not all rights can be obtained, the operation fails;
- The operation executes, reading and writing the objects of the database;
- Used rights are released, except when consumed (escrow); new rights may also be created in this step.

After the local execution, the side effects of the operation in the data and reservation objects are propagated and executed in other replicas asynchronously and atomically.

Note that reservations guarantee that operations that can lead to invariant violation do not execute concurrently, but they do not guarantee that the preconditions for the operation to generate side effects hold. For example, in the tournament, before removing a tournament it is necessary to disenroll all players, to ensure that no player in enrolled. These properties have to be checked by the application.

Reservations manager The reservations manager is a service that runs in each datacenter and is responsible for exchanging reservations between datacenters, tracking reservations in use by local clients, and providing clients the database snapshot information to access the underlying storage. For correctness, it is necessary to enforce that updates of an operation are atomic and that reads are causally consistent with the current rights at each replica. In Indigo, these properties are guaranteed directly by the underlying storage system.

An example shows why these properties are necessary. In our tournament application, to enroll a player it is necessary to obtain the right that allows the enroll operation to execute (by forbidding the removal of both the player and the tournament). After the enroll completes, the right is released and can be obtained by an operation that wants to remove the tournament. The problem is that if the state observed by the remove tournament operation did not include the previous enrollment, the application could end up deleting the tournament even though a players has enrolled in it, leading to an invariant violation.

Obtaining reservation rights The first and last phases of operation's execution is to obtain and free the rights needed for executing that operation. Indigo provides API functions for obtaining and releasing a list of rights. Indigo tries to obtain the necessary rights locally using ordered locking to avoid deadlocks. If other datacenters need to be contacted for obtaining some reservation rights, this process is executed before starting to obtain rights locally. Unlike the process for obtaining rights in the local datacenter, Indigo tries to obtain the needed rights from remote datacenters in parallel for minimizing latency. This approach is prone to deadlocks; therefore, if some remote right cannot be obtained, we use an exponential backoff approach that frees all rights and tries to obtain them again after an increasing amount of time.

When it is necessary to contact other datacenters to obtain some right, the latency of operation execution can be severely affected. Therefore, reservation rights are obtained proactively using the following strategy. Multi-level lock and multi-level mask rights are pre-allocated to allow executing the most common operations (based on the expected frequency of operations), with shared allow and forbid rights being shared among all datacenters. Escrow lock rights are divided among datacenters, with a datacenter asking for additional rights to the datacenter it believes has more rights (based on local information). The primary of an escrow lock for conditions creates new rights by computing the number of missing rights whenever either it runs out of rights or the object is marked for reevaluation. In the tournament example, *shared forbid* for removing tournaments and players can be owned in all datacenters, allowing the more frequent enroll operation to execute locally. Partition lock rights are initially assigned to a single replica, and transferred when needed.

The reservations manager maintains a cache of reservation objects and allows concurrent operations to use the same shared (allow or forbid) right. While some ongoing operation is using a shared or exclusive right, the right cannot be revoked. The information about ongoing operations is maintained in soft-state. If the machine where the reservations manager runs fails, the ongoing operation will fail when trying to release the obtained rights.

6.2.3 Fault tolerance

Indigo builds on the fault tolerance of the underlying storage system. In a typical geo-replicated store, data is replicated inside a datacenter using quorums or a state-machine replication algorithm. Thus, the failure of a machine inside a datacenter does not lead to any data loss. This also applies to the machine running the reservations manager: as explained before, ongoing transactions will fail in this case; committed changes to the reservation objects are stored in the underlying storage system.

If a datacenter (fails or) gets partitioned from other datacenters, it is impossible to transfer rights from and to the partitioned datacenter. In each partition, operations that only require rights available in the partition can execute normally. Operations requiring

rights not available in the partition will fail. When the partition is repaired (or the datacenter recovers with its state intact), normal operation is resumed.

In the event that a datacenter fails losing its internal state, the rights held by that datacenter are lost. As reservation objects maintain the rights held by all replicas, the procedure to recover the rights lost by the datacenter failure is greatly simplified: it is only necessary to guarantee that recovery is executed only once with a state that reflects all updates received from the failed datacenter.

6.3 Evaluation

This section presents an evaluation of Indigo. The main question our evaluation tries to answer is how does *explicit consistency* compares against *causal consistency* and *strong consistency* in terms of latency and throughput with different workloads. Additionally, we try to answer the following questions:

- Can the algorithm for detecting I-offender sets be used with realistic applications?
- What is the impact of increasing the amount of contention in objects and reservations?
- What is the impact of using an increasing number of reservations in each operation?
- What is the behavior when coordination is necessary for obtaining reservations?

6.3.1 Applications

For evaluating Indigo we implemented two applications that are representative of real-world services. These applications allowed us to evaluate the global performance of the approach, and to micro-benchmark the

Ad counter The ad counter application models the information maintained by a system that manages ad impressions in online applications. This information needs to be geo-replicated for allowing the fast delivery of ads. For maximizing revenue, an ad should be impressed exactly the number of times the advertiser is willing to pay for. This invariant can be easily expressed as $nrImpressions(A_i) \leq K_i$, where K_i is the maximum number of times ad A_i should be impressed and the function $nrImpressions(A_i)$ returns the number of times it has been impressed.

Advertisers will typically require ads to be impressed a minimum number of times in some countries. For instance, ad A should be impressed exactly 10,000 times, with at least 4,000 impressions in the US and another 4,000 impressions in the EU. This example is modeled through the following invariants for specifying the limits on the number of impressions (where *nrImpressionsOther* counts the sum of the number of impressions in

datacenters other than those two with the impressions in excess of 4,000 in the EU or the US):

$$nrImpressionsEU(A) \leq 4,000$$
$$nrImpressionsUS(A) \leq 4,000$$
$$nrImpressionsOther(A) \leq 2,000$$

We modeled this application by having one counter for each ad and region pair. Invariants were defined with the target limits stored in the database: $nrImpressions(R, A) \leq targetImpressions(R, A)$ A single update operation that increments the ad tally was defined, which increments the function *nrImpressions*. Our analysis shows that two increment operations for the same counter can lead to an invariant violation, but increments on different counters are independent. Invariants can be enforced by relying on escrow lock reservations for each ad.

Our experiments used workloads with a mix of: a read-only operation that returns the value of a set of counters selected randomly; an operation that reads and increments a randomly selected counter. Our default workload included only increment operations.

Tournament management This is a version of the application for managing tournaments described in Section 5 (and used throughout this chapter as running example), extended with read operations for browsing tournaments. The operations defined in this application are similar to operations that one would find in other management applications such as courseware management.

As detailed throughout this and the previous chapter, this application has a rich set of invariants, including uniqueness rules for assigning ids; generic referential integrity rules for enrollments; and numeric invariants for specifying the capacity of each tournament. This leads to a reservation system that uses both escrow lock for conditions and multi-level lock reservation objects. There are three operations that do not require any right to execute: add player, add tournament and disenroll tournament, although the latter accesses the escrow lock object associated with the capacity of the tournament. The other update operations involve acquiring rights before they can execute.

In our experiments we have run a workload with 82% of read operations (a value similar to the TPC-W shopping workload), 4% of update operations requiring no rights for executing, and 14% of update operations requiring rights (8% of the operations are enrollment and disenrolments).

6.3.2 Experimental Setup

We compare Indigo against three alternative approaches:

Causal Consistency (Causal) As our system was built on top of the causally-consistent SwiftCloud system [130], we have used unmodified SwiftCloud as representative of a system providing causal consistency. We note that this system cannot enforce invariants. This comparison allows us to measure the overhead introduced by Indigo.

Strong Consistency (Strong) We have emulated a strongly consistent system by running Indigo in a single DC and forwarding all operations to that DC. We note that this approach allows more concurrency than a typical strong consistency system as it allows updates on the same objects to proceed concurrently and be merged if they do not violate invariants.

RedBlue consistency (RedBlue) We have emulated a system with RedBlue consistency [83] by running Indigo in all DCs and having red operations (those that may violate invariants and require reservations) execute in a master DC, while blue operations execute in the closest DC, while respecting causal dependencies.

Our experiments comprised 3 Amazon EC2 datacenters, US-East, US-West and EU, with inter-datacenter latency presented in Table 6.2. In each DC, Indigo servers run in a single m3.xlarge virtual machine with 4 vCPUs and 8 ECUs of computational power, and 15GB of memory available. Clients that issue transactions run in up to three m3.xlarge machines. Where appropriate, we placed the master DC in the US-East datacenter to minimize the overall communication latency and this way optimize the performance of the configurations that require cross-replica coordination.

RTT (ms)	US-E	US-W
US-West	81	–
EU	93	161

Table 6.2: RTT Latency among datacenters in Amazon EC2.

6.3.3 Latency and Throughput

We start by comparing the latency and throughput of Indigo with alternative deployments for both applications.

We ran the ad counter application with 1000 ads and a single invariant for each ad. The maximum number of impressions was set sufficiently high to guarantee that the limit is not reached. The workload included only update operations for incrementing the counter. This allowed us to measure the peak throughput when operations were able to obtain reservations in advance. The results are presented in Figure 6.1, and show that Indigo achieves throughput and latency similar to a causally consistent system. Strong and RedBlue results are similar to each other, as all update operations are red and execute in the master DC in both configurations.

Figure 6.2 presents the results when running the tournament application with the default workload. As before, results show that Indigo achieves throughput and latency similar to a causally consistent system. In this case, as most operations are either read-only, or can be classified as blue, the throughput of RedBlue is only slightly worse than Indigo.

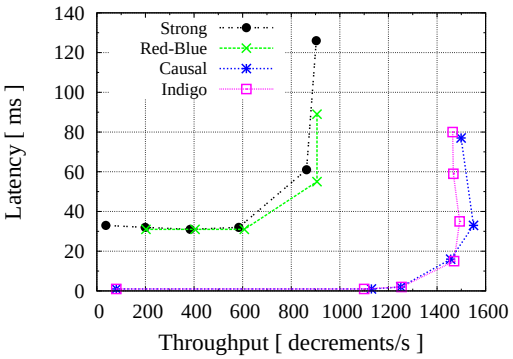


Figure 6.1: Peak throughput (ad counter application).

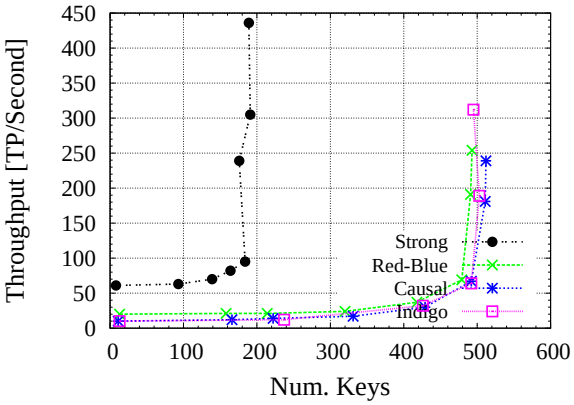


Figure 6.2: Peak throughput (tournament application).

Figure 6.4 details these results, presenting the latency per operation type (for selected operations) in a run with throughput close to the peak value. The results show that Indigo exhibits lower latency than RedBlue for red operations. These operations can execute in the local DC in Indigo, as they require either no reservation or reservations that can be shared and are typically locally available.

Two other observations that deserve some discussion: *Remove tournament* requires canceling shared forbid rights acquired by other DCs before being able to acquire the shared allow right for removing the tournament, which explain the high latency. Sometimes latency is very high (as shown by the line with the maximum value). This is a result of the permissions exchange algorithm being asynchronous, and because requesting remote DCs to cancel their rights incurs in delays when rights are being used.

Add player has a surprisingly high latency in all configurations. Analyzing the situation, we found out that the reason for this lies in the fact that this operation manipulates very large objects used to maintain indexes, causing all configurations to have a fixed overhead.

6.3.4 Micro-benchmarks

Next, we examine the impact of key parameters in the performance of the system.

Increasing contention Figure 6.3 shows the throughput of the system with increasing contention in the ad counter application, by varying the number of counters in the experiment. As expected, the throughput of Indigo decreases when contention increases as several steps require executing operations sequentially. Furthermore, the results reflect the fact that our middleware introduces an additional level of contention, because operations have to contact the reservation manager.

Increasing number of invariants Figure 6.6 presents the results of the ad counter application with an increasing number of invariants involved in each operation: the operation

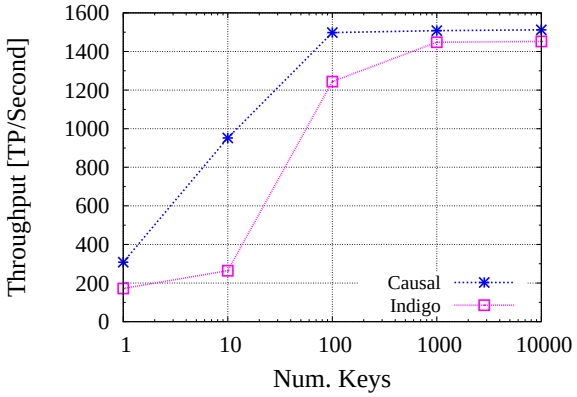


Figure 6.3: Peak throughput with increasing contention (ad counter application).

reads 5 counters (R5) and updates one to three counters (W1 to W3). In this case, the results show that the peak throughput for Indigo decreases while latency keeps constant. The reason for this is that for escrow locks, each invariant has an associated reservation object. Thus, when increasing the number of invariants, the number of updated objects also increases, with an impact on the operations that each datacenter needs to execute. To verify our explanation, we ran a workload with operations that access the same number of counters in the weak consistency configuration. The presented results show the same pattern of decreased throughput.

Impact when transferring reservations Figure 6.5 shows the latency of individual operations executed in the US-W datacenter in the ad counter application, for a workload where increments reach the invariant limit for multiple counters and where the rights were initially assigned to a single datacenter. When rights do not exist locally, Indigo cannot mask the latency imposed by coordination, in this case, for obtaining additional rights from the remote datacenters. This explains the high latency operations close to the start of the experiment. As a bulk of rights is obtained, the following operations execute with low latency until it is necessary to obtain additional rights. When a replica believes that no other replica has available rights in an escrow lock object, it does not contact replicas. Instead, the operation fail locally, leading to low latency.

In Figure 6.4, we showed the impact of obtaining a multi-level lock shared right that requires revoking rights present in all other replicas. We already discussed this problem. Nevertheless, it is important to note that such impact in latency is only experienced when it is necessary to revoke shared forbid rights in all replicas before acquiring the needed shared allow right. The positive consequence of this approach is that enroll operations requiring the shared forbid right that was shared by all replicas can execute with latency close to zero. The maximum latency line in enroll operation shows the maximum latency experienced when a replica acquires a shared forbid right from a replica that holds such right.

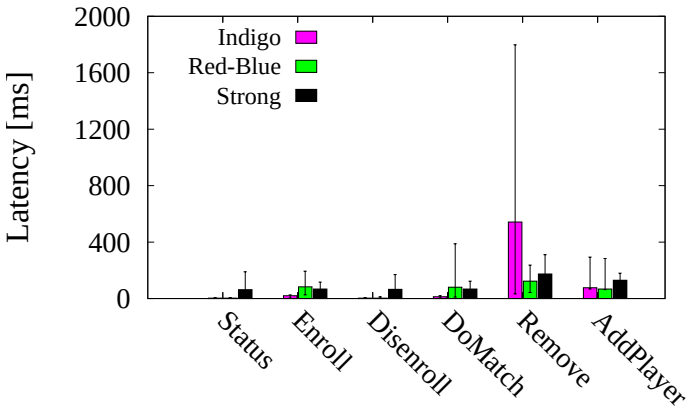


Figure 6.4: Average latency per op. type - Indigo (tournament application).

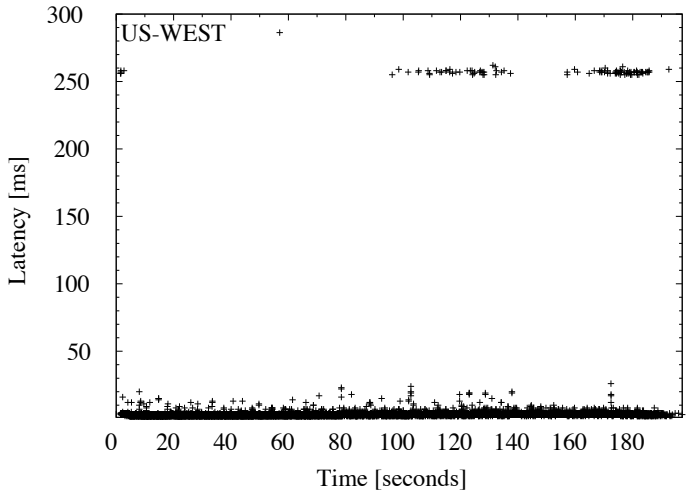


Figure 6.5: Latency of individual operations of US-W datacenter (ad counter application).

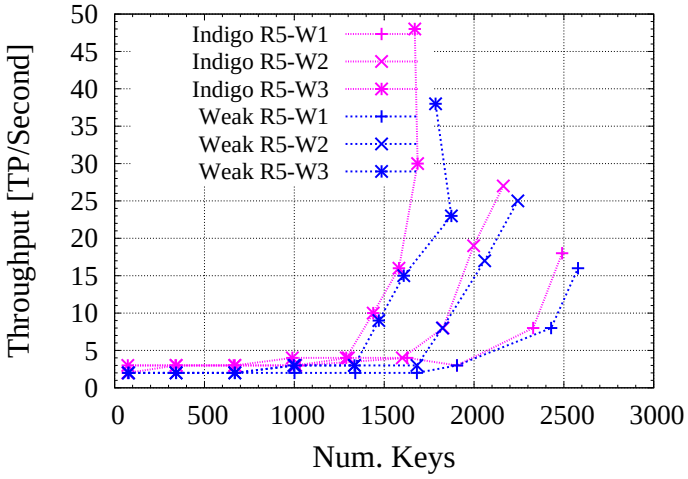


Figure 6.6: Peak throughput with an increasing number of invariants (ad counter application).

6.4 Related work

Many cloud storage systems supporting geo-replication emerged in recent years. They provide different sets of features that try to satisfy the requirements of different applications, without compromising performance, latency, or availability.

Some approaches allow reading a causally consistent view of the database (causal consistency) [6, 9, 44, 86]; others support limited transactions where a set of updates are made visible atomically [12, 87]; or support application-specific or type-specific reconciliation with no lost updates [22, 40, 86, 117], etc. Other approaches leverage the semantics of applications to go a step further [50, 63, 77, 83, 112, 117]. Semantic types [50] have been used for building non serializable schedules that preserve consistency in distributed databases. Conflict-free replicated data types [112] explore commutativity for enabling the automatic merge of concurrent updates, which Walter [117], Gemini [83] and SwiftCloud [130] use as the basis for providing eventual consistency. However, none of those approaches enable general invariant preservation under weak consistency.

Concurrently to our work, the homeostasis protocol [110] has been proposed. The objective of this work is to ensure strong consistency while minimizing coordination between nodes. The techniques that the authors employ are similar to ours: they use a static analysis to extract information from the transactions in the workload and use a mechanism similar to the demarcation protocol to execute those parts of the code without coordination when it is safe. Contrarily to our approach, the authors use a special language for specifying the transactions that can be optimized. As a positive aspect is that this analysis can be automatized, removing the programmer from the loop. The downside, is that the code analysis is very difficult, thus the types of operations that can be optimized are limited. During execution, the homeostasis may use coordination in order to adapt the limits of each replica. In contrast, our solution relies exclusively on peer-to-peer communication to enhance availability and partition-tolerance.

Indigo shares the same principle of distinguishing categories of operations as red-blue consistency [83]. The main difference between the two approaches is that Indigo is able to execute some operations that are classified as conflicting concurrently with other operations that do not conflict with some instances of those operations, improving the latency for those operations in the general case.

6.5 Final remarks

In this chapter we have explored the idea of using reservations to enforce application invariants without compromising the latency and availability of applications. Reservations allow to control the concurrent execution of operations at a fine grain, without requiring replicas to communicate with each other to execute each operation.

We propose new reservations data types that are suitable to prevent invariant violations that are common in applications. The new designs are suitable for geo-replicated

environments and allow replicas to share reservations for executing operations that are non-conflicting when executed with certain parameters. Moreover, the multi-level lock design that we have presented is a general mechanism that can be used to solve any I-offender set (the proof is given in [55]). We provide new data-type designs that, similarly to the Bounded Counter, depend only on features that are available in existing storage systems to work correctly (in this case the system additionally needs to provide a transactional interface).

We have developed Indigo, a prototype of a system that provides reservations on top of existing key-value stores. Indigo automatically manages reservations in a geo-replicated deployment, exchanging reservations on demand, when reservations are missing, and proactively for provisioning. We demonstrated the usability of our approach by implementing applications representative of real-life services.

The evaluation of the system shows that, overall, Indigo exhibits low latency for executing most operations, however, some operations might have higher latency, when it is necessary to revoke reservations over multiple replicas. This effect could be mitigated by implementing efficient algorithms for provisioning reservations or using sporadic coordination to ensure that reservations are exchanged fast. The investigation of these techniques is an open subject.

A downside of this approach is that distributing reservations over multiple replicas might make the system unavailable for executing certain operations, if the replica goes down without freeing the reservations. To circumvent this issue, either replicas expose uncertainty to the client, to make him aware that the operation might still abort, or use compensations to resolve any mistake that is detected later.

In the next chapter we pursue an alternative route for achieving explicit consistency that removes coordination completely from applications. In some cases we are able to modify operations to make them conflict-free, without altering their semantics, in other cases we need to resort to compensations to fix any mistakes.

CHAPTER 7

IPA

Minimizing the use of coordination for executing operations improves the general performance and availability of applications. However, for achieving true high availability and fault tolerance, ideally we would like to forego coordination completely.

In this chapter, we explore an alternative route for achieving explicit consistency that does not use cross-replica coordination. The idea is to modify the logic of operations, at development time, to prevent invariant violations due to concurrent executions. Since writing such logic is a complex and application-dependent task, we propose *IPA*, an algorithm for modifying operations in a way that meets this property. For each conflicting pair of operations, the *IPA* algorithm transforms those operations and check if the generated operations are conflict-free. We use CRDT convergence policies to transform operations, and rely on the conflict detection mechanism that was introduced in chapter 5 to check if the modified operations are non-conflicting.

We extended the conflict detection algorithm to make use of convergent data types. In the previous version of the algorithm, if two operations set two different values for the same predicate, the algorithm would signal those operations as conflicting. The extended version of the algorithm allows to specify convergence policies for each data type. During the analysis, or beforehand, the programmer inputs the convergence policy for each predicate that has opposing values, and the algorithm evaluates if the operation is still conflicting after applying the converge rule. We observed that picking the right converge rules allows to solve many conflicting operations without further modifying operations.

Second, when we need to modify operations, we try to modify them in a way that prevents operations from making the resulting database state incompatible with other concurrent operations. To give an example, imagine that we are adding a player to a tournament. The local operation checks if the tournament exists, adds the player to

the tournament and propagates the effects to the remaining replicas. If a concurrent operation at a remote replica removes that tournament, and the system does not check if the pre-conditions of the enroll operation are true when applying its effects (i.e. that the tournament exists), the resulting state will be invalid. To prevent this, it is necessary to modify the enroll or the remove operations in a way that both operations can always be applied without breaking the invariant. One solution is to re-create the tournament during the enroll operation, to ensure that it appears again, even if it is removed by a concurrent operation. Alternatively, we can modify the remove operation to automatically cancel any concurrent enrollments for that tournament.

The modifications to the operations can be made preventively, by modifying the specification of operations, or they can be applied when conflicts actually occur during execution, using compensations [51].

It is up to the programmer to apply transformations preventively or not. In some situations, despite being able to maintain the invariant, applying the effects preventively might result in bad user experience. Using compensations can provide better semantics in some cases. For instance, to apologize for undoable actions, or to allow the user to pick the result that fits better in a certain situation. On top of that, the IPA algorithm can be used alongside Indigo, providing a full-range of solutions for maintaining invariants that can be used to achieve good user experience and performance.

We designed a tool that integrates the algorithm for modifying operations with the algorithm for detecting conflicts, allowing programmer to solve conflicting pairs of operations by choosing alternative specifications generated by *IPA*. We studied a list of invariants expressed in real web applications [13] and found that, in many cases, it is possible to transform operations to become invariant-preserving, while maintaining a semantics that is acceptable from the standpoint of the application logic. To support these transformations, we had to design new CRDTs, new convergence policies and add support for compensations in some CRDT designs.

Our evaluation shows that the proposed approach leads to latency and scalability similar to the baseline of weak consistency, while preserving global application invariants. When compared to Indigo, *IPA* provides more predictable guarantees, as no operations experience high latency due to the need of coordinating with other replicas.

This chapter is organized as follows: in section 7.1 we explain how to transform applications to make them conflict-free; in section 7.2 we present the algorithm for generating new operation specifications; in section 7.3 we discuss the implementation of the tool and the new data type designs; in section 7.4 we evaluate the practicability of the approach, the types of invariants that it covers, and evaluate the performance of transformed applications; and, finally, we conclude in section 7.6.

7.1 Invariant preserving applications

The typical logic of operations consists in checking certain logical conditions and produce a set of effects based on the validity of those conditions. For example, when enrolling a player in a tournament, the application first checks that the player and the tournament exist. If, and only if, both conditions are true, the operation executes producing effects.

Under strong consistency, the effects of operations are applied in equivalent database states for all replicas. However, under weak consistency, the state of remote replicas might be modified concurrently between the execution of the operation at the origin and the application of the effects, therefore, the value of the pre-conditions might be different when the operation is applied remotely. In some cases this does not pose any problem to the application. For instance, if some other players enroll in the tournament concurrently, this does not affect the correctness of the application (assuming there is no limit on the number of players for the tournament). But, in other cases, applying the effects of operation might make the state of the database inconsistent, for instance if the player or the tournament were removed.

7.1.1 Adding effects to operations

Our observation is that in many cases applications can be implemented correctly on top of weak consistency by augmenting operations with additional effects that prevent that the effects of the application break the invariant of the database when applied in remote replicas. Doing this might result in canceling the effects of some of the conflicting operations, however, programmers can decide which semantics they prefer, if a conflict of this nature occurs.

Going back to the example, the effects of the $enroll(p,t)$ operations violate the pre-condition of the $rem\_tournament(t)$ if both operations execute concurrently, because the latter operations requires that no player is enrolled in tournament t, while the former adds a new player to that tournament. Similarly, the effects of $rem\_tournament(t)$ violates the pre-condition of $enroll(p,t)$, which requires that the tournament exists after enrolling the player in the tournament. To solve the conflict between the two operations, the $enroll(p,t)$ can be augmented with an effect to recreate the tournament, or the $rem\_tournament(t)$ can be augmented to ensure that all players that are enrolled concurrently in tournament t are removed. Applying the additional effects before the effects of the original operation guarantee that the local state will satisfy the pre-conditions of the modified operation.

7.1.2 Applying convergence policies

Besides adding additional effects to enforce the sufficient pre-conditions of operations, it is necessary to choose convergence policies carefully for each object to ensure that concurrent updates do not modify the intended value for the object. For instance, when modifying the $enroll(p,t)$ operation it is necessary to use an *Add-wins* policy to ensure that

the tournament t is not eliminated by the effects of *rem_tournament*(t) (which removes the tournament). Note that adding these extra effects to the operations does not affect the perceived semantics of the operation when executed in stand-alone, but that they provide a precise semantics when executed concurrently with other conflicting operations.

We have verified that following this method for modifying operations helps to ensure many common invariants without requiring coordination. However it is a difficult task to be carried by programmers, because it requires that programmers are knowledgeable of convergence policies to ensure correctness.

IPA is a tool to automate the process of generating alternative operation specifications that are non-conflicting. The algorithm does that by testing modifications to each pair of conflicting operations, taking into account the convergence semantics of the data types in place. The generated operations are modified and testes for conflicts, iteratively, until they do not conflict with any other operation in the workload. To search for possible alternative specifications of operations, the algorithm only uses predicates from invariant expressions that were violated in the conflict. This strategy allows to reduce the search space and preserve the original semantics of the operation, in the general case.

For operations that do not have possible valid modifications, the operation is signaled and an alternative mechanism for preventing conflicts must be used.

7.2 IPA methodology

For building an invariant-preserving version of a given application, the programmer must accomplish the following steps:

Step 1: specify application: The first step consists in building a specification of the application by identifying application invariants and operation effects. Inferring this information automatically is outside the scope of this work [84, 110].

Step 2: generate invariant-preserving specification: *IPA* iteratively proposes modifications to the application, until all operations are non-conflicting. First, the algorithm picks a pair of conflicting operations, if any. Next, a list of possible modifications to make the pair safe under concurrency is presented to the programmer. In general, each resolution strategy will have the effects of one operation prevail over the effects of the other. The programmer is required to choose which resolution provides the semantics that better suits the application. If no suitable modifications exist for some conflicting pair, the unresolved conflict is flagged. The algorithm repeats until all conflicts are resolved or flagged.

Step 3: Modify application: The output of the previous step is an updated specification of the application, stating the conflict-resolution associated with each predicate and the effects of each operation. The programmer can then patch the original application according to the new specification, adding the necessary effects, which typically requires only a few additional lines of code, as detailed in Section 7.4.1.3. For conflicts flagged as

unsolvable by *IPA*, the programmer can resort to some coordination mechanism to avoid concurrent execution of the offending operations [16, 83].

Fully patched applications can execute in any replicated system that provides causal consistency, highly available transactions and the necessary type-specific conflict resolution policies. A number of systems support these features [4, 117, 130].

We implemented the *IPA* tool as a proof-of-the-concept of the proposed methodology. Programmers interact with the tool during the analysis process to choose the preferred resolution rules for each data-type and the preferred resolutions for conflicting operations. The tool is capable of transforming an extensive collection of non *I-Confluent* operations, such as general boolean expressions, referential integrity and some numerical invariants.

7.2.1 Making operations invariant-preserving

In this section we present the algorithm in more detail. In listing 7.1 we present the specification of the application that was presented in section 5.2.2 for convenience.

```
1  @Unique("player(p)")
2  @Unique("tournament(t)")
3  @Inv("forall(Player:p, Tournament:t) :- enrolled(p,t) =>
4   player(p) and tournament(t)")
5  @Inv("forall(Player:p,q, Tournament:t) :- inMatch(p,q,t) =>
6   enrolled(p,t) and enrolled(q,t) and (active(t) or finished(t))")
7  @Inv("forall(T : t) :- nrPlayers(t) <= Capacity")
8  @Inv("forall(T : t) :- active(t)  => nrPlayers(t) >= 1")
9  @Inv("forall(Tournament:t) :- active(t) => tournament(t)")
10 @Inv("forall(Tournament:t) :- finished(t) => tournament(t)")
11 @Inv("forall(Tournament:t) :- not( active(t) and finished(t))")i
12 public interface TournamentApp {
13
14   @True("player(p)")
15   RESULT addPlayer(Player p);
16
17   @True("tournament(t)")
18   RESULT addTournament(Tournament t);
19
20   @False("tournament(t)")
21   RESULT remTournament(Tournament t);
22
23   @True("enrolled(p, t)")
24   @Increments("nrPlayers($1, 1)")
25   RESULT enroll(Player p, Tournament t);
26
27   @False("enrolled(p, t)")
28   @Decrements("nrPlayers($1, 1)")
29   RESULT disenroll(Player p, Tournament t);
30
31   @True("active(t)")
32   RESULT beginTournament(Tournament t);
33
34   @True("finished(t)")
35   @False("active(t)")
36   RESULT finishTournament(Tournament t);
37
38   @True("inMatch(p,q,t)")
39   RESULT doMatch(Player p, Player q, Tournament t);
40 }
```

Listing 7.1: Specification of the tournament management system written in Java.

Algorithm 8 presents the logic for creating an invariant-preserving version of an application. We define this algorithm as a function that receives as input the invariant, I, the set of operations, *Ops*, and a set of convergence rules, *CR*, defined for each predicate by the programmer. The algorithm only handles boolean predicates (lines 1 to 5); in Section 7.2.5 we explain how to extend the algorithm to support numeric invariants.

The main loop iterates over all pairs of conflicting operations until no more conflicts exist. For each conflicting pair (line 3), the algorithm replaces the initial operation's

Algorithm 8 *IPA* algorithm and main functions.

```
    ▷ IPA main loop.
 1: function IPA(I, Ops, CR)
 2:     while existsConflictingPair(I, Ops, CR) do
 3:         opPair ← findConflictingPair(I, Ops, CR)
 4:         newPair ← repairConflicts(I, opPair, CR)
 5:         Ops.replace(opPair, newPair)
        return Ops
    ▷ Extended conflict detection algorithm.
 6: function isConflicting(I, OpPair, CR)
 7:     if opposingEffects(OpPair) then
 8:         newOpPair ← apply(OpPair, CR) return checkConflicting(I, newOpPair, CR)
 9:     else return checkConflicting(I, OpPair, CR)
    ▷ IPA algorithm for repairing conflicts.
10: function repairConflicts(I, OpPair, CR)
11:     sols ← ∅
12:     invPreds ← {getPreds(i) | i ∈ invClauses(I, opPair)}
13:     newOpPairs ← generate(invPreds, I, OpPair)
14:     for opPair ∈ newOpPairs do
15:         if not isPairSubset(opPair, sols) then
16:             if not isConflicting(I, opPair, CR)) then
17:                 sols ← sols ∪ { opPair }
        return pickResolution(sols)
    ▷ New operation generation.
18: function generate(invPreds, I, (op1, op2))
19:     seed ← {p(true) ∪ p(false) | p ∈ invPreds}
20:     effectSets ← powerSet(seed)
21:     pairs ← ∅
22:     for p1 ∈ effectSets do
23:         pairs ← pairs ∪{(newOp(op1,p1),op2)}
24:         pairs ← pairs ∪{(op1,newOp(op2,p1))}
        return order(pairs)                    ▷ by increasing no. of predicates.
```

specification (line 5) with the new specification that solves the identified conflict (line 4). If there are no alternative safe operations for the conflicting pair with a given set of convergence rules, the pair is flagged as unsolvable and the algorithm continues, ignoring that pair in subsequent iterations.

7.2.2 Conflict detection

Figure 7.1 shows schematically the conflict analysis for operations *rem_tournament*(t) and *enroll*(p, t). The algorithm determines the weakest preconditions for executing both operations: the predicates tournament t and player p must be *true*. This condition is inferred automatically from the application invariant. In practice, the application code might never produce a state that violates this pre-condition, however, checking the weakest precondition is sufficient to maintain correctness [55].

From S_{init}, the execution of each operation individually leads to S_1 and S_2 respectively. Merging both states (or applying the effect generated in one replica in the state of the other replica), leads to S_{final}, which is an invalid database state because the player is enrolled in a tournament that does not exist anymore. The state obtained from the execution of

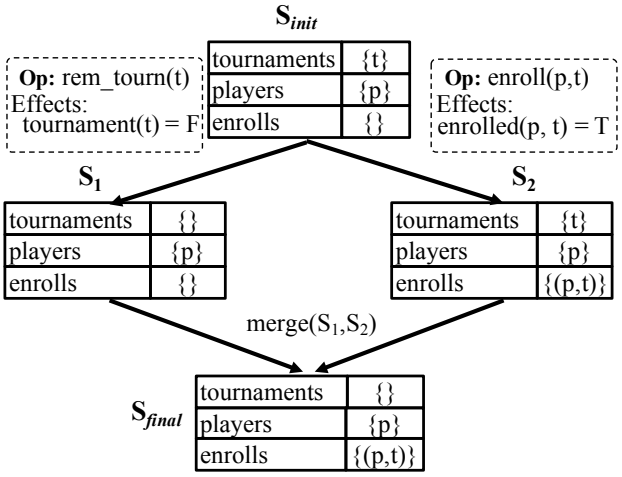


Figure 7.1: Example of an execution breaking referential integrity.

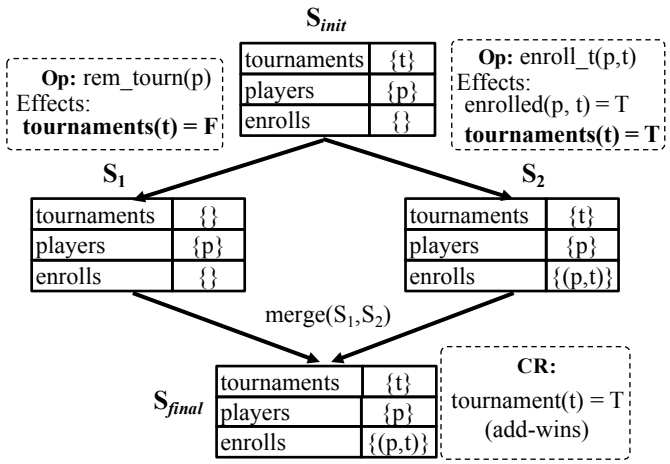


Figure 7.2: Execution with modified *enroll*(p,t) operation. Invariant is preserved.

each operations is incompatible with the pre-conditions of the other operation, but this is no longer checked by the local replica before applying the effects of the operation. The algorithm marks the pair of operations as conflicting.

The conflict in this operations occurs due to the modification of the predicate tournament. The *enroll*(p,t) operation assumes that this predicate is true, while the *rem_tournament*(t) operation sets its value to false. To fix this conflict, one option is to force the value of that predicate to be true when applying the operation *enroll*(p,t).

We extended the conflict detection algorithm presented in chapter 5 to support the use of convergence policies during conflict evaluation. Contrarily to the previous version of the algorithm, when assigning two different values for the same predicate, a convergence policy, $r \in CR$, can be applied to force a specific value for a predicate that was modified by the two operations with different values. A convergence rule r can specify that the final value for a predicate that is modified concurrently is *true* for an *Add-wins* policy and

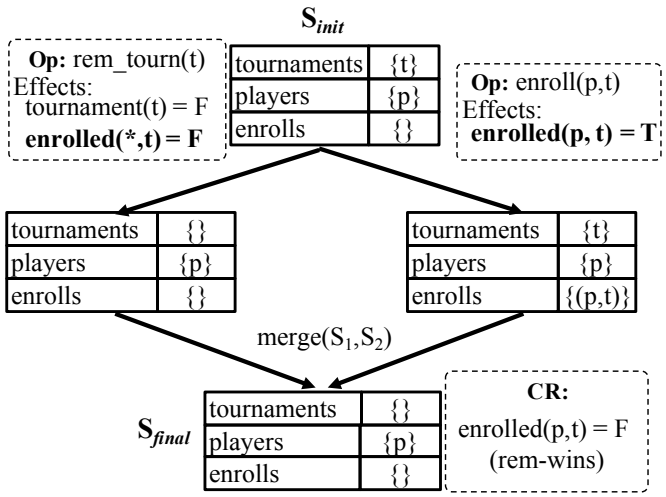


Figure 7.3: Execution with modified *rem_tournament*(t) operation. Invariant is preserved.

false for a *Rem-wins* policy. Supporting convergence policies in the analysis is crucial to ensure the pre-conditions when applying operations concurrently. We use this technique to solve the conflict presented before, as explained in the next section.

Function *isConflicting* (line 6) presents the conflict detection algorithm. In line 7, the function checks if the operations have opposing effects on at least one predicate. If so, the algorithm replaces the predicate value in each operation with the values from the convergence policies in *CR*. Then, it checks whether the combined effects of the operations may break the invariant (line 8).

7.2.3 Proposing modified operations

Our algorithm heuristically identifies the set of effects that need to be added to each operation to guarantee that replicas always converge to a correct state. In the example, the violation can be repaired either by giving preference to the effects of *rem_tournament*(t) or *enroll*(p,t). In the former case, it is necessary to guarantee that no player enrolls in t concurrently with a *rem_tournament*(t); in the latter case, that tournament t is not removed concurrently with a *enroll*(p,t).

Function *repairConflicts* (line 10) starts by selecting, for a conflicting pair, the invariant clauses that might be involved in the conflict (namely the invariant clauses that have predicates affected by the effects of the operations), and creates a pool of predicates for generating new operations (line 12). The next step of the algorithm is to heuristically generate new operations with combinations of those predicates (line 13). Line 15 checks if the new operations are not included in any previous solution, ensuring that the number of predicates added to the generated operations is minimal. Next, the algorithm tests if the new operations solve the conflict that was identified (line 16). All operations that solve a conflict are stored and one of them is chosen as the resolution for the conflict, which can be done either manually or according to some policy (line 17).

The modified operations solve the conflict between the pair of operations, but they might still conflict with other operations. Successive iterations of the algorithm will then fix all remaining conflicts (as said before, any unsolvable conflict is detected and flagged).

The *generate* function (line 18) computes all possible combinations of effects that can be added to each operation. The algorithm computes the powerset of predicates in *invPreds*, with different predicate values *true* and *false*, and adds each element of the set to each operation, ignoring any predicates that are already present in the operation. The function only modifies one operation in each pair (lines 23 and 24). The generated operations are ordered by the number of predicates (line 24) to ensure that the algorithm suggests modified operations with fewer predicates first (in line 16).

7.2.4 Example

In this section, we analyze the modified operations proposed by the *IPA* algorithm, using the example of the *rem_tournament*(t) and *enroll*(p,t) conflict. The invariant violated by the concurrent execution of both operations is the following: $I = enrolled(p,t) \Rightarrow player(p) \wedge tournament(t)$. The algorithm uses these predicates to generate new sets of effects for the operations. We show how to modify each operation to preserve its effects over the effects of its counterpart.

Figure 7.2 shows operation *enroll*(p,t) with an added effect to set the predicate *tournament*(t) to *true*, which makes the operation non conflicting with *rem_tournament*(t). When this operations executes against operation *rem_tournament*(t) and an *Add-wins* policy is used for predicate tournament, tournament t is recreated, as shown in the final state of the figure. We note that the additional effect has no impact if there is no concurrent *rem_tournament*(t), as the tournament has to exist for enroll to be executed. This modification gives preference to the *enroll*(p,t) over *rem_tournament*(t), with the effects of the first operation prevailing while the effects of the latter are undone.

An alternative resolution, depicted in Figure 7.3, consists in giving preference to *rem_tournament*(t) by guaranteeing that the final database state has no player enrolled in tournament t. This can be achieved by setting the predicate *enrolled*$(*,t)$ to *false* and using a *Rem-wins* policy. The wildcard $(*)$ specifies that the predicate applies to any player – this is necessary since it is impossible to know beforehand which players might enroll in tournament t. With the additional effect, an *enroll*(p,t) will have no effect when executed concurrently with a *rem_tournament*(t). In section 7.3.1 we describe how to implement the effect with a wildcard efficiently.

After selecting a resolution for the conflict, the algorithm proceeds by checking if the new operations conflict with any other operations. For instance, similar conflicts appear when considering the pair *enroll*(p,t) and *rem_player*(p). Our algorithm composes the resolution of multiple operations together, until solving all conflicts.

If the programmer is not satisfied with a set of solutions proposed by the algorithm, he might provide a different conflict resolution set to search for alternative operation

specifications. In our prototype, the programmer is not forced to specify the convergence rule for all predicates. Instead, whenever the algorithm finds a modification that conflicts with some predicate that does not have an assigned convergence policy, the tool asks the programmer to choose one at that moment.

7.2.5 Compensations

Some invariant violations cannot be prevented beforehand with a reasonable semantics. Consider as an example the constraint in listing 7.1, line 5, that enforces a maximum number of players enrolled in a tournament. To prevent this violation it would be necessary to remove a player from the tournament whenever a player is added, to ensure that the size is always within bounds, however, doing this would make the application unusable. In this case, we only want to disenroll a player if the size of the tournament is exceeded.

Instead of applying extra effects on every operation execution, the system can delay applying the extra effects to a later point in time, and only do that, if a violation actually occurs. This mechanism is known as compensation [26, 51, 62, 98, 122]. *IPA* can also generate compensations for restoring pre-conditions.

The analysis can automatically generate compensations for certain constraints, like aggregation and numeric constraints, that only execute if an invariant violations is detected during execution. During the analysis, instead of adding the new effects to one of the conflicting operations, the algorithm creates a new operation with the effects, and simulates the execution of that operation alongside the conflicting pair, to check if that operation corrects any invariant violation. In this case, conflicts might still occur during execution because the conflicting operations are not modified. Therefore, to guarantee that applications are correct at all times, any operation that accesses a predicate involved in one of these constraints (which can be inferred from the analysis) must check if the constraint is valid. If a violation occurred, the compensation is executed before exposing the value to the client, to ensure that the database state is repaired.

We provide data types that encapsulate the constraint checking and compensations execution out-of-the-box, which are automatically triggered when the object is accessed. We explain the implementation of those data types in Section 7.3.1.

Our approach has the benefit of being totally decentralized, allowing replicas to detect and apply compensations without coordination. The downside is that if conflicts are detected simultaneously by different replicas, and each replica has observed divergent database states, they might apply different compensations. This does not affect convergence, because compensations in *IPA* are designed to be commutative, idempotent and monotonic, however the semantic for programmers and clients might be worse (for instance, cancel two flight reservations, while only one ticket was oversold).

7.3 Implementation

This section describes the implementation details of this work. We mainly focus on the design of the new data types, the algorithm, and tools. The reference platform is based on the work presented in chapter 6.

7.3.1 CRDTs for supporting IPA

We now discuss the CRDTs used for implementing the conflict-resolutions used in *IPA*.

7.3.1.1 Specialized convergence policies

As discussed in previous sections, we rely on CRDT convergence policies to enforce invariant maintenance. The SwiftCloud system includes an extensive library of CRDTs that can be used to support many *IPA* transformations. Nonetheless, we had to extend the existing set data type with a variation of the *Rem-wins* convergence policy to support predicates with wildcards, such as $enrolled(*, t) = false$. The effect of this predicate is to clear the elements of the set, which can be used to ensure that the tournament is empty. However, the *Rem-wins* set can only ensure that elements that are removed from the set will not be added, thus it does not prevent adding an element to the set that has not been seen before.

We provide a new data type design, the *Resetable Remove-wins Set*, that has a *reset*() operation that removes all existing elements from the set and prevents any concurrent addition. To support this semantics, we extend the existing *Rem-wins* set with a vector clock that tracks the last time the set was pruned with a *reset*() call. Every *add*() operation sends the associated vector clock (i.e. the current version of the object at the moment the operation was issued), and if it is concurrent or smaller than the current pruning vector clock, the element is not added to the set. A specification for this data type can be seen in algorithm 9.

7.3.1.2 Compensation CRDTs

For some invariants, it is possible to encapsulate the logic for detecting conflicts and repairing the state of the object automatically in the data types. For example, consider a restriction that enforces that there is a maximum number of players in a tournament. Concurrent operations might break this invariants. However, it can be restored by removing any exceeding players from the tournament when any players accesses the tournament.

To ensure that the application is always consistent, whenever an operations reads the set storing the players of the tournament, it is necessary to check that the size of the set is within limits, and, in case it is not, it is necessary to remove the exceeding elements.

We provide a Set CRDT that limits the number of elements in the set. When the size limit of the set exceeds, the exceeding elements are removed automatically from the set,

Algorithm 9 Operation-based Resetable Remove-Wins Set.

```
1:  payload Set E, R, V
2:      initial ∅,∅,∅                ▷ E Set of element, R Set of pairs (element, unique_id), V Vector Clock
3:  query contains (element e) : boolean b
4:      b = (e ∈ E) ∧ not(∃_uid : (e, uid) ∈ R)
5:  update add (element e)
6:      prepare(e)
7:          D = {(e, uid)|∃_uid : (e, uid) ∈ R}
8:          v = current_clock()
9:      downstream(e, v, D)
10:             R := R \ D                          ▷ Removes all visible uids.
11:         if v ≥ V then                           ▷ Ensures concurrent elements are not revived.
12:             E := E ∪ {e}
13: update remove (element e)
14:     prepare(e)
15:         uid = unique()
16:     downstream(e, uid)
17:         R := R ∪ {(e, uid)}
18: update reset ()
19:     prepare(e)
20:         v = inc_get_current_clock()             ▷ Increment and get the current clock value.
21:     downstream(v)
22:         V := merge(v, V)
23:         E := ∅
```

and a compensation function is applied for each of those elements to ensure that the state of the database remains consistent after the compensation.

Contrarily to the escrow transactional method, CRDTs with compensations do not preclude operations from executing. In exchange, the data type must ensure that the invariant that they enforce holds every time the object is read. To that end, every operation of the data type is guarded by a method that checks if the invariant is valid at that moment, and in case it is not, it applies the compensation immediately to restore that condition. Compensation are executed locally at the replica that detects the invariant violation and applied in all replicas, as any other operation of the data type. The downside of this approach, is that different replicas might observe different replica's state when applying compensations, which might result in applying compensations more times than necessary. However, since operations execute at all replicas, the resulting state of the object will be correct in respect to the invariant.

The specification in algorithm 10 shows the specification of the *Limited-Size Add-Wins Set*. In this data type, each element is associated with a vector clock, which indicates the time at which it was created. Every operation of the data type is guarded by the function *enforceContraint*(), which restores the invariant in case it has been broken by any concurrent operations. To restore the invariant, the *enforceContraint*() removes elements from the set using a compensation operation. To select the element for removal, it determines the set of elements that have an higher vectors clock and were added concurrently, and select one of them randomly for removal. Alternatively, we could order elements according to some rule to ensure that if two replicas see the same subset of operations,

Algorithm 10 Limited size add-wins set.

```
1:  payload Set S, Integer L, Function C
2:      initial ∅, L, C   ▷ Set of pairs (element, unique_id), size limit of the set, and compensation function
3:  update contains (element e)boolean b
4:      prepare(e)
5:          enforceContraint()
6:          b = (∃_uid : (e, uid) ∈ S)
7:  update add (element e)
8:      prepare(e)
9:          enforceContraint()
10:         if #{(e|(e, uid) ∈ S} ≤ L then
11:             v = inc_get_current_clock()        ▷ Increment and get the current clock value.
12:     downstream(e, v)
13:         S := S ∪ {(e, v)}
14: update remove (element e)
15:     prepare(e)
16:         enforceContraint()
17:         if contains(e) then
18:             R = {(e, v)|∃_v : (e, v) ∈ S}
19:     downstream(R)
20:         S := S \ R                             ▷ Removes pairs identified at source.
21: function enforceConstraint()
22:     while #{(e|(e, v) ∈ S} > L do
23:         (e, v_e) = random_concurrent()         ▷ Select an element for removal.
24:         remove(e)
25:         C(e)
```

they will remove the same element. For each removed element, the data type executes a user-provided function C to apply any side effects that are necessary (e.g. notify the player that he has been disenrolled from the tournament).

It is possible to design other CRDTs with built-in repairable invariants using the same approach.

7.4 Evaluation

In this section, we present an evaluation of *IPA*, meant to answer the following questions:

(i) Which invariants are covered by our approach?

(ii) What is the effort of using *IPA*?

(iii) How does the performance of applications modified by *IPA* compare to other solutions that use coordination to maintain invariants in detriment of performance, or do not maintain invariants?

7.4.1 Invariant preservation with *IPA*

This section surveys the invariants covered by our approach by analyzing the use of *IPA* in several applications. The objective of this study is to understand what types of invariants are covered by our method. For classes invariants that cannot be maintained under weak consistency by modifying application specifications, some form of coordination is

Inv. Type	*I*-Conf.	*IPA*	TPC	Tour	Ticket	Twitter
Sequential id.	No	No	Yes	—	—	—
Unique id.	Yes	Yes	Yes	Yes	Yes	Yes
Numeric inv.	No	Comp.	Yes	—	—	—
Aggreg. const.	No	Comp.	—	—	—	—
Aggreg. incl.	Yes	Yes	—	Yes	—	—
Ref. integrity	No	Yes	Yes	Yes	—	Yes
Disjunctions	No	Yes	—	Yes	—	—

Table 7.1: Types of Invariants present in applications.

required. Indigo can be used in that situation at the trade of higher latency for certain operations and loss of availability in certain cases.

7.4.1.1 Classes of invariants

Prior work has analyzed the invariants that are used in real applications [11, 13, 83]. In that study, the authors characterized which classes of invariants can be implemented correctly under weak consistency (i.e. that are *I-Confluent*). We build Table 7.1 based on that study, summarizing which classes of invariants can be preserved by implementing operation without any further modifications to the code (column *I-Confluent*) and which invariants can be maintained by transforming applications to work correctly under weak consistency.

Sequential identifiers: Sequential identifiers are useful for establishing an unique total order of elements. In general, generating these identifiers requires coordination to avoid collisions. No solution, based on weak consistency can maintain this invariant. However, it has been shown that, in most cases, applications could easily replace the use of sequential identifiers by unique identifiers [10, 125].

Unique identifiers: Unique identifiers can be preserved without coordination at runtime. It suffices to pre-partition the space of identifiers among the nodes that will generate them to avoid collisions.

Numeric invariants: Numeric invariants assert conditions involving numeric predicates (e.g., $p < k$). In general, preserving these invariants requires coordination. However, support is possible on top of weak consistency by relying on escrow techniques [15, 66, 93]. In *IPA*, we can use compensations to preserve this type of invariants, whenever the semantics is reasonable for the application [26]. For example, to replenish the stock of a product, like in TPC-C/W.

Aggregation constraint: Imposing a bound on the size of a collection, e.g., limiting the players enrolled in a tournament, can be addressed using a numeric invariant over a predicate that represents the size of the collection, thus sharing its properties.

Aggregation inclusion: Ensuring an element is eventually added or removed from a collection is *I-Confluent*, provided no dependencies to other objects exist. If that is not the case, then preserving referential integrity is usually required.

Referential integrity: Preserving relations and dependencies among objects, such as

foreign keys in relational databases and references to keys in key-value stores, is not *I-Confluent*. *IPA* fully supports this invariant, as exemplified throughout this chapter.

Disjunctions: Applications often specify that one of several conditions must be met by using a disjunction. *IPA* addresses this type of invariant by extending an operation to ensure that the disjunction is always true. This is an extension of the mechanism for supporting referential integrity, as in this case there might be several alternative conditions that restore the validity of the invariant.

7.4.1.2 Invariants in applications

We now analyze how *IPA* can address the invariants that are present in several representative applications.

IPATournament This application showcases some of the invariants that our solution can address. For this application, *IPA* is capable of proposing multiple alternative resolutions that either reconstruct broken dependencies, or clear them, to avoid inconsistencies due to concurrent executions, as discussed throughout this chapter. We do not impose a limit to the number of enrolled players in the implementation. That type of constraint is evaluated in a separate application.

IPATwitter We implemented a clone of Twitter that relies heavily on referential integrity to implement user timelines and maintain subscribers information. When some user tweets, we opted for writing immediately to all followers timelines. This emphasizes consistency issues that arise when tweets or users are removed concurrently. Our version explores several alternatives for solving these conflicts. If a tweet is retweeted and removed concurrently, the options are to recover the deleted tweet or hide all of its retweets from the followers timelines. As for handling user removals, *IPA* can leverage the *Rem-wins* semantics to purge all the user's history from the timelines of the other users concurrently with any other operations that might be happening.

***IPATicket*:** this application is based on FusionTicket [49, 66, 128]. The main invariant of this application is that tickets for events cannot be oversold. It is necessary to use compensations in this case, as it is impossible to prevent the violation beforehand, as discussed in 7.2.5. When the tickets available are oversold, the application cancels the ticket and reimburses the user with temporary balance within the applications. The transference of money to the client's account crosses the boundaries of the system thus it needs to use a different mechanism to ensure that the operation is not lost and completes correctly.

***TPC-W* and *TPC-C*:** These standard database benchmarks overlook some aspects of real-world applications, such as having operations to manage product listings. In our specification, we extended these applications to include such operations, which introduced referential integrity constraints. For addressing the lack of inventory after purchase, we used *IPA* compensations to increase the stock (as in the specification of the benchmark). An alternative would be to cancel the oversold purchases, as in the previous example.

7.4.1.3 Using the *IPA* tool

The *IPA* algorithm generates new operation specifications by testing conflicts and augmenting operations, in an iterative process, supervised by the programmer. The number of tests that our tool generates is bounded by the number of operations in the application specification. Again, we rely on the Z3 SMT solver [39] to test all valid combinations of parameters efficiently. Despite the satisfiability problem having exponential complexity, the solver is capable of handling most cases in polynomial time. In our tests, using a modern laptop, this automatic step of the algorithm was fast enough to not hinder interactivity and frustrate the programmer.

```
void ensureEnroll(String p, String t) {
 AddWinsSet tournamentIndex = getCRDT(TOUR_IDX, TYPE_OF_AW_SET);
 AddWinsSet playerIndex = getCRDT(PLR_IDX, TYPE_OF_AW_SET);
 tournamentIndex.add(t);
 playerIndex.add(p);
}

void ensureDoMatch(String p1, String p2, String t) {
 ensureEnroll(p1, t);
 ensureEnroll(p2, t);
}

void ensureBegin(String t) {
 AddWinsSet tournamentIndex = getCRDT(TOUR_IDX, TYPE_OF_AW_SET);
 tournamentIndex.add(t);
}

void ensureEnd(String t) {
 AddWinsSet tournamentIndex = getCRDT(TOUR_IDX, TYPE_OF_AW_SET);
 RemWinsSet tStarted = getCRDT(T_STARTED_IDX, TYPE_OF_RW_SET);
 tournamentIndex.add(t);
 tStarted.remove(t);
}
```

Listing 7.2: Additional code to make operations conflict-free in the *IPATournament* application.

In terms of the work required to write the modified version of the application, this effort is small. For example, listing 7.2 presents the code of the auxiliary functions that are necessary to restore the consistency of the *IPATournament* application. These functions are executed alongside with the corresponding operations to ensure that the invariants of the application are restored remotely, in case some conflicting operation executes concurrently. The other applications that we have implemented follow a similar scheme. Only a few lines of code are necessary to add to each conflicting operation.

Listing 7.3 shows the compensation for the oversold flight tickets in the *IPATicket* application. This compensation is registered in the object that stores the ticket reservations and it is triggered for every element that is automatically removed from the set.

```
int sizeLimit = CAPACITY;
LimitedSet flightReservations = getCRDT(FLIGHT_INDEX, TYPE_OF_LS_SET);
flightReservations.setHandler(new CompensationHandler(sizeLimit) {
    public void execute(String removedCostumer) {
  SetCRDT costumerReservations;
  costumerReservations = getCostumerReservations(removedCostumer);
  costumerReservations.remove(flightId);
  }
 });
```

Listing 7.3: Compensations for oversold flight tickets in *IPATicket* application.

7.4.1.4 Discussion

The invariants that the *IPA* tool can support are limited to the extent of invariants that can be expressed using the language that we have defined. The classes of invariants that we support (table 7.1) are common in many Internet application, as remarked by Bailis et al. [13]. The examples discussed in the previous section show that the language is expressive enough to address rather complex applications, including applications based on relational databases.

If a database is shared by multiple applications, the programmer must create a single specification of all applications for the analysis to identify all possible conflicts. The alternative would be to provide the resolution mechanisms at the storage level and to repair invariants independently of the applications developed on top. We chose to apply transformations at application level to show the possibility of implementing the applications without changing the underlying storage.

The effort of writing specifications is arguably comparable to the effort of writing the code itself [100]. A lot of research has dealt with this problem, proposing automatic feature extraction, and code synthesis, to aid the programmer in writing correct applications [8, 46, 48, 84, 110]. Our approach stands to benefit from these complementary research avenues.

7.4.2 Performance evaluation

In this section, we compare the performance of modified applications against other solutions. We expect the modifications to have a minimal overhead in comparison to the original code running on weak consistency. We also expect the latency of the operations to be clearly lower in comparison to systems that use coordination to enforce invariant preservation. We also try to measure the tipping point at which solutions based on coordination are faster than executing extra updates. For this, we use synthetic benchmarks.

7.4.2.1 System configurations

The benchmarks execute in a geo-replicated setting on Amazon EC2. The database deployment consists of three servers running in three geographical regions, with average latency around 80 milliseconds between US-EAST and US-WEST and US-EAST and EU-WEST, and 160 between EU-WEST and US-WEST.

The application server is co-located with the storage system deployed in each region. We use SwiftCloud to implement all different approaches that we evaluate. Clients are installed in other machines in the same availability zones as the corresponding closest servers.

We compare the performance of applications with the following configurations:
Causal Consistency (*Causal*) Unmodified applications, does not maintain invariants for conflicting operations.

Inv. Preserving Applications (*IPA*) Applications modified using *IPA*, maintains invariants on top of *Causal*.

Strong Consistency (*Strong*) all update operations are forwarded to a single server to enforce serialization. We use the US-EAST replica to execute updates and to minimize the average latency.

Invariant violation avoidance (*Indigo*) Applications modified with coordination mechanisms to prevent conflicting operations from executing concurrently. We use Indigo for implementing this configuration. For this experiments we improved the reservations manager of Indigo, to give priority to reservations that are less popular, to reduce the high latency that was observed for certain operations in our previous experiments.

7.4.2.2 Throughput and latency

We evaluate the scalability of each configuration of the system by measuring the latency of operations with different loads on the system, using the *IPATournament* application. 35% of the operations in the workload execute writes, and each operation might execute multiple reads and writes in the context of an interactive transaction, thus the time for executing each operation might vary. All write operations are conflicting in the original specification and in the modified version, all operations are *I-Confluent* and use a mix of conflict resolution policies. In the version used for *Indigo*, all write operations need to acquire some reservation. Operation *status* reads information about tournaments. The operations is read only and thus non-conflicting. It is the dominant operation in the workload.

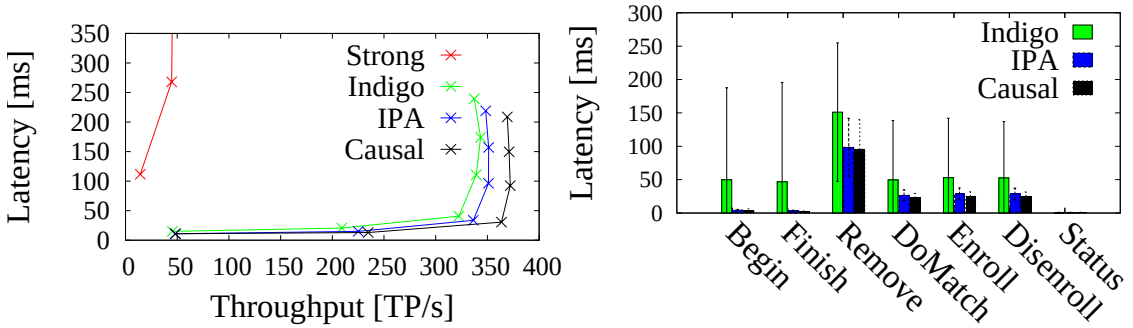
To test the scalability of the system, we increase the number of clients contacting each server by running extra client threads until peak throughput is achieved. Figure 7.4a shows the latency of operations for each number of requests per second.

The results show that *Strong* presents the highest average latency, which is a consequence of having $\frac{2}{3}$ of operations being forwarded to a remote server and being executed in sequence. Despite that there are existing strong consistency solutions that scale better than our approach [35], the base latency of those approaches is comparable to ours. The variations in the time for executing operations are explained by the different number of reads and writes each operation performs.

Causal shows the best scalability with the lowest latency. Our approach, *IPA*, performs slightly worse than *Causal*, as additional updates need to be executed, but enforces application invariants. When compared to *Indigo*, our approach performs slightly better. The advantage is small because, while each operation requires acquiring a reservation, reservations are exchanged among replicas very infrequently after that.

Figure 7.4b presents the latency for the write operation and highlights more clearly the differences between the configurations. We omit the strong consistency column. The average latency of operations in *Indigo* is higher than the latency for *IPA* or *Causal*. This is explained by the occasional need for *Indigo* replicas to trade reservations. Compared

to *Causal*, the latency of the write operations is only slightly higher in the *IPA* approach, which is due to the extra code they execute. The overhead of executing extra effects is discussed in Section 7.4.2.5.



a Peak throughput b Latency of individual operations

Figure 7.4: Performance of *IPATournament* using different approached.

7.4.2.3 Comparing different strategies

We implemented *IPATwitter* using *Add-wins* and *Rem-wins* strategies to compare the overheads of each approach. Figure 7.5 shows the latency of each operation for the different strategies. The *Add-wins* version must ensure that when a user tweets, or retweets, he cannot be removed concurrently. This incurs in the cost of restoring the user for those operations and explains their higher latency compared to *Causal*. Whereas, *Rem-wins* strategy must ensure that a tweet does not appear in any user's timeline if that tweet is removed concurrently. Pessimistically, this would have to remove the tweet from the timelines of every user in the system, as the tweet could be added to anyone's timeline (via a re-tweet). Instead, we enact this strategy with a compensation, applied when accessing user timelines. This hides tweets that were removed, thus restoring the invariant, trading a slightly higher latency in reads to prevent unnecessary writes.

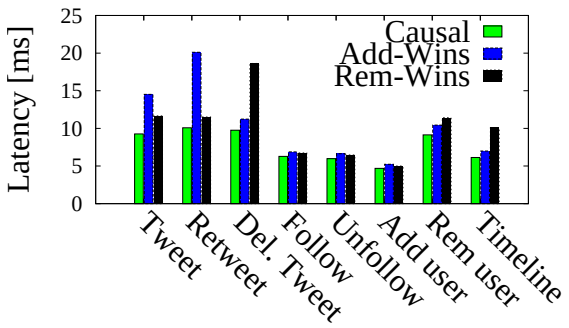


Figure 7.5: Latency of individual operations in *IPATwitter*.

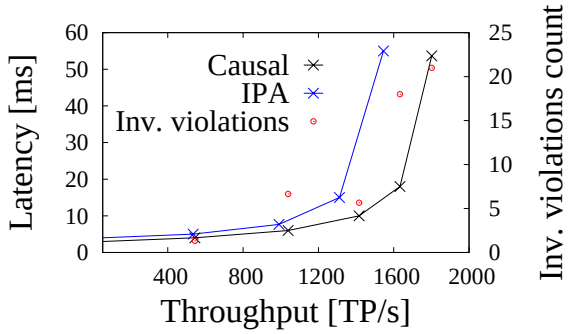


Figure 7.6: Peak throughput for *IPATicket* benchmark. Red dots indicate number of invariant violations observed in *Causal*.

7.4.2.4 Compensations scalability

We evaluate the scalability of the compensations CRDT in the *IPATicket* application, by increasing contention. Figure 7.6 presents the latency of operations for a certain load of the server. The small dots in the figure indicate the average number of invariant violations that were observed at that throughput, when using *Causal*. They confirm the intuition that as contention rises, the divergence window grows larger, increasing the chance for invariant violation. In *Causal*, this exposes the application consistency anomalies, while in *IPA* executing the compensations preserves the invariants at all times. As expected, compensations incur on some overhead, but still provide latency comparable to *Causal*.

7.4.2.5 Microbenchmarks

IPA avoids invariant violations by adding extra updates to one or multiple objects in an operation. In this section, we evaluate the overhead of adding additional effects to operations. We analyze the impact of executing increasingly more updates in a transaction in comparison to the costs of executing the original operation in strong consistency or Indigo.

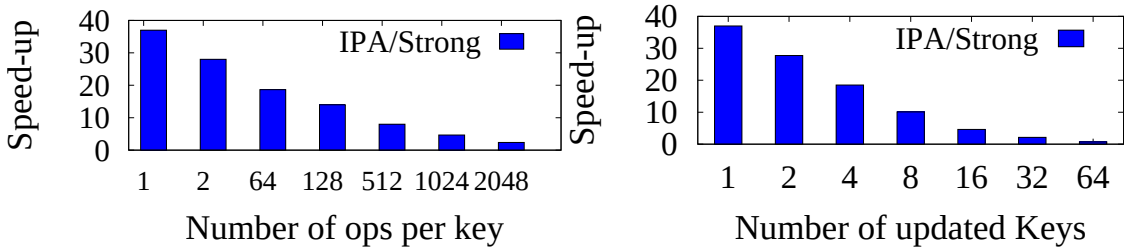
Operations on a single object: We measure the speedup of an application running on top of causal consistency that executes extra updates for a single object versus the original operation running on *Strong*. Figure 7.7a shows that the original operation is about 28× faster in *IPA* than in *Strong*. Adding more updates to this operation makes the speedup decrease, because the transaction takes more time to execute. When we execute 2048 updates to a single object, the average latency is still about 40ms. Executing updates on a single object imposes low extra overhead on the system, because the object is read and written to storage only once and subsequent updates only impose processing costs.

Operations over multiple objects: Now we evaluate the overhead when an operations executes multiple updates over different keys. In this case we expect the penalty of modifying operations to increase faster because it is necessary to fetch more objects from storage for a single operation.

The original application reads a varying number of objects to check some condition

and then executes a single write operation to an object. The modified application checks the same condition, but executes a write for each object. The idea is to simulate the number of extra updates that an operation might need to execute. Figure 7.7b shows performance dropping faster than when executing updates over single objects, as expected. At 64 objects, it starts to pay off to use *Strong*.

In practice, in the applications that we evaluated, we require only a few extra updates per object for a small number of objects, which support that is reasonable to modify operations this way. In the case of *IPATwitter*, which needs to execute more writes due to the way the timeline is implemented, we were able to execute them lazily via compensations, thus avoiding the high latency penalty.



a Speed-up operations per key.

b Speed-up operations number of keys.

Figure 7.7: Speed-up of executing multiple writes in IPA versus unmodified *Strong*.
Speed-up of executing multiple writes in IPA versus unmodified *Strong*.

Comparison with *Indigo*: In Indigo, operations are allowed to execute locally if the replica holds some specific reservations. Multiple operations might be able to execute concurrently at different replicas if all of them can share the same reservations. If a replica requires changing the value of some reservation that is being used, it must request remote replicas to release it, before acquiring it. This approach only avoids coordination when a replica holds the necessary reservations to execute some operation. Thus, the latency of an application depends on the contention for obtaining the reservations.

In this experiment, we evaluate the impact of varying the percentage of operations that compete to acquire some reservations. We compare the performance of this solution against executing the same operation in *IPA*. Figure 7.8 shows that *IPA* performance is equivalent to *Indigo* with no contention for reservations, and that the latency of *Indigo* rises steadily as contention increases.

Despite the overhead for executing the additional effects, *IPA* provides a predictable latency for operations, which is not the case for *Indigo*, whose operations latency depend on the current distribution of reservations. Furthermore, our approach is fault-tolerant as a client can execute operations as long as it can access a single server. In *Indigo*, if a server that holds the necessary reservation to execute some operation becomes unavailable, the operation cannot be executed.

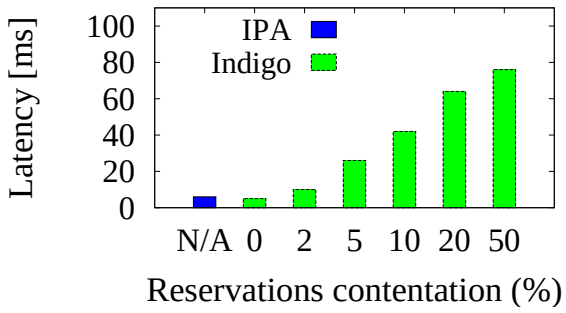


Figure 7.8: Latency of operations in comparison to reservations.

7.5 Related work

Systems that use weak consistency are widely deployed in the real-world [22, 31, 32, 78, 116], but can be difficult to use correctly [13].

To mitigate the problem, RedBlue [83] and Walter [117] provide support for executing operations under weak or strong consistency to allow fast operations when invariants are not at risk, and resort to strong consistency when the execution of operations is unsafe [83, 117]. Despite improving the latency of operations in the general case, systems that depend on coordination may still become unavailable and exhibit high latency.

Convergent data types [112] provide automatic conflict resolution rules, which lessens the programming effort to ensure replica convergence. However application entities might have inter-object dependencies that constraint the convergence policies that are can be used for each entity. The *IPA* tool can be used to help programmers to choose convergence policies appropriately when building complex applications using weak consistency, to ensure invariant preservation without requiring coordination.

Bayou [122] ensures invariant preservation without requiring coordination by forcing programmers to specify conflict detection and resolution mechanisms, and re-executing some operations. Conversely, *IPA* does not need to run conflict detection or re-execute operations for most types of invariants, as the modifications done to operations can be executed preventively.

However, for some types of invariants, modifying the effects of operations make their semantics poor. We provide support for compensations [51, 62, 98] to fix invariant violations when they cannot be prevented beforehand. In our work, we generate the effects of compensations automatically and design new data types that are capable of automatically check for violations, to reduce the effort for the programmer to apply this technique.

The execution of compensations usually requires some support from the underlying system to ensure state convergence [51, 122]. In our approach, we leverage convergent data types to execute them as any other operation of the system. This strategy allows that different replicas solve the same invariant violation in different ways (depending on the replica's state), which ultimately might lead to applying unnecessary (but correct) effects. However, it allows compensations to be used without any extra support from the system.

7.6 Final remarks

In this chapter we explored a different route for achieving explicit consistency that does not require coordination in any way. Instead of coordinating the execution of conflicting operations, we have experimented the possibility of redefining operations effects to preserve invariants at all times. We showed that our technique is able to correct a variety of invariant violations common in applications without changing the semantics of operations when executed in standalone. We proposed an automated process for generating new specifications and proving that operations are *I-Confluent*, relieving programmer from that effort.

To extend the range of invariants that we could cover, we also added support for compensations, which allow to defer the application of effects to repair invariant violations after the fact. This is necessary in some cases, as adding the effects to the operations would result in a unsatisfactory semantics for applications, which could render the applications unusable.

Experimental results back the viability of the approach, showing that the modified applications present a performance similar to the original applications. The features required from the underlying storage system are available in several existing weakly consistent databases, which facilitates the implementation of this technique of top of several systems.

Chapter 8

Conclusion

In this chapter we conclude with final remarks about the work that was carried and briefly discuss future research directions that arise from our findings.

In this thesis we have studied the design of applications on top of geo-replicated storage. The engineering challenge in this topic is to provide good service properties, such as availability and low latency at a global scale, while maintaining applications consistent.

Our approach consists in exploring application's semantics to enforce stronger consistency in applications, while keeping systems available under partitioning. We propose a novel consistency model, explicit consistency, that defines consistency using the application invariants, instead of constraining the execution order of operations, as traditionally done. This different characterization of consistency allows us to achieve a better balance between guaranteeing application correctness and ensuring low latency and high availability.

We provide a methodology for implementing explicit consistency that allows to identify potential invariant violations that may result from concurrent executions. Based on that information we explored two complementary approaches for preventing those violations.

Violation avoidance: The first approach consists in leveraging the information about conflicts to introduce points of synchronization that prevent possible conflicting executions. The same idea has been explored in previous work [33, 83, 84], but only with strict delimitation between operations that are safe and unsafe. Our contribution is the possibility of deciding dynamically if it is necessary to coordinate a particular operation execution to maintain correctness, or if the operations can execute asynchronously, based on the current state of the system. This allows to reduce the use of synchronization when

compared to other systems, ensuring general good availability and low latency while maintaining invariants.

We implemented this approach, first by extending an existing key-value store to support maintaining numerical invariants, and later, we generalized it for systems that support transactions, making the approach usable in a wider range of applications.

The downside of this approach is that, even if synchronization is only used sporadically, the application will unavoidably lose availability for executions that require contacting remote replicas. The invariant preservation approach does not suffer from this limitation.

Invariant preservation: The second approach takes a more exploratory vision. We have observed that many invariant violations can be prevented by defining a precise result for the execution of conflicting operations. In many cases, invariant violations can be prevented by adding some extra effects to operations to ensure that the state of the application remains consistent under concurrency. The advantage of this approach is that is not necessary to use synchronization for preventing conflicts, which results in lower latency and higher availability for applications.

For operations that cannot be modified without affecting the perceived semantics of operations, we propose the use of compensations. Compensations allow to detect and fix invariant violations after the fact. Compensations can be executed before clients are able to query the application's state, ensuring that it is repaired before being exposed to the clients.

We proposed an algorithm that is capable of transforming operations specifications and deriving compensations for specific operations, allowing programmers to achieve highly-available applications.

The approach shows that in some cases it is possible to make a small trade in the semantics of operations for higher availability, while only affecting the semantics of operations when conflicts occur. If the semantics of operations is not viable for a certain application, the programmer has always the option of coordinating the execution of those operations, which we have showed that can be done very efficiently.

The tools: Both approaches come with companion tools that aid programmers to transform applications. The tools serve as proof-of-concept of the proposed methodology. The tools pinpoint problematic operations, while the programmer is responsible for modifying operations in a way that prevents conflicts. While we recognize that these tools are far from being usable in practice, they are an important milestone in the way of making a full-fledged methodology for developing correct-by-design distributed application.

To conclude, the final contribution of this thesis is a deeper insight on the types of invariant violations that occur in weakly consistent systems. While the trade-off remains a difficult problem for programmers, our work provides well-founded solution that can be applied based on reasonable assumptions that can be made on nowadays systems.

More precisely, our approach only requires that the underlying system provides causal consistency and supports transactions.

8.1 Research directions

In this section we discuss potential directions for the outcomes of this work

Complex applications: In this work we have narrowed the scope of an application to stand-alone applications on top of a single database. However the more general case, is that applications are built with multiple services that are composed with each other and may depend on data that is scattered and duplicated across multiple databases. This makes the case of ensuring application-wide invariants much more challenging, because the consistency properties across services of these systems are much weaker.

One possible solution for that problem is to develop a methodology that allows programmers to model the interactions between multiple cooperating services, and do an analysis to identify the potential conflicts that might occur for operations that execute across services.

Usability: Requiring programmers to provide a specification of applications is a barrier for the adoption of our methodology. It is difficult to specify applications using a formal logic, even if it is simple as first-order logic, and even more difficult to ensure that applications meet the specification.

A possible direction for this problem is to incorporate the mechanisms used in each approach in a language that is already familiar for programmers, such as SQL. We are already studying the hypothesis of modifying an SQL engine to support reservations and automatic conflict-resolution. A second alternative, is to extract the specification directly from the code, with minimum input from the programmer, to automatically analyze and transform applications. This would allow a more systematic and less error-prone usage of the methodology.

Bibliography

[1] M. Abadi and L. Lamport. "The Existence of Refinement Mappings". In: *Theor. Comput. Sci.* 82.2 (1991), pp. 253–284. URL: http://dx.doi.org/10.1016/0304-3975(91)90224-P.

[2] A. Adya. "Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions". AAI0800775. PhD thesis. Cambridge, MA, USA, 1999.

[3] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. "Sinfonia: A New Paradigm for Building Scalable Distributed Systems". In: *SIGOPS Oper. Syst. Rev.* 41.6 (2007), pp. 159–174. URL: http://doi.acm.org/10.1145/1323293.1294278.

[4] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. "Cure: Strong Semantics Meets High Availability and Low Latency". In: *Proocedings of the IEEE 36th International Conference on Distributed Computing Systems*. ICDCS '16. 2016, pp. 405–414.

[5] P. S. Almeida, A. Shoker, and C. Baquero. "Delta State Replicated Data Types". In: *CoRR* abs/1603.01529 (2016). URL: http://arxiv.org/abs/1603.01529.

[6] S. Almeida, J. a. Leitão, and L. Rodrigues. "ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 85–98. URL: http://doi.acm.org/10.1145/2465351.2465361.

[7] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. "Consistency Analysis in Bloom: a CALM and Collected Approach". In: *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*. CIDR '11. Asilomar, CA, USA, 2011, pp. 249–260. URL: http://cidrdb.org/cidr2011/Papers/CIDR11_Paper35.pdf.

[8] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. "Blazes: Coordination Analysis for Distributed Programs". In: *Proceedings of the IEEE 30th International Conference on Data Engineering*. ICDE '14. Chicago, IL, USA, 2014, pp. 52–63. URL: http://dx.doi.org/10.1109/ICDE.2014.6816639.

[9] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. "Bolt-on Causal Consistency". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: ACM, 2013, pp. 761–772. URL: http://doi.acm.org/10.1145/2463676.2465279.

[10] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. "Highly Available Transactions: Virtues and Limitations". In: *Proc. VLDB Endow.* 7.3 (2013), pp. 181–192. URL: http://dx.doi.org/10.14778/2732232.2732237.

[11] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. "Coordination Avoidance in Database Systems". In: *Proc. VLDB Endow.* 8.3 (2014), pp. 185–196. URL: http://dx.doi.org/10.14778/2735508.2735509.

[12] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. "Scalable Atomic Visibility with RAMP Transactions". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 27–38. URL: http://doi.acm.org/10.1145/2588555.2588562.

[13] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. "Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1327–1342. URL: http://doi.acm.org/10.1145/2723372.2737784.

[14] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services". In: *Proceedings of the Conference on Innovative Data system Research*. CIDR '11. 2011, pp. 223–234. URL: http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf.

[15] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. M. Preguiça. "Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants". In: *Proceedings of the 34th IEEE Symposium on Reliable Distributed Systems*. SRDS '15. Montreal, QC, Canada, 2015, pp. 31–36. URL: http://dx.doi.org/10.1109/SRDS.2015.32.

[16] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. "Putting Consistency Back into Eventual Consistency". In: *Proceedings of the 10th European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015, 6:1–6:16. URL: http://doi.acm.org/10.1145/2741948.2741972.

[17] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. M. Preguiça, M. Najafzadeh, and M. Shapiro. "Towards Fast Invariant Preservation in Geo-replicated Systems". In: *Operating Systems Review* 49.1 (2015), pp. 121–125. URL: http://doi.acm.org/10.1145/2723872.2723889.

[18] V. Balegas, C. Li, M. Najafzadeh, D. Porto, A. Clement, S. Duarte, C. Ferreira, J. Gehrke, J. Leitão, N. M. Preguiça, R. Rodrigues, M. Shapiro, and V. Vafeiadis. "Geo-Replication: Fast If Possible, Consistent If Necessary". In: *IEEE Data Eng. Bulletin* 39.1 (2016), pp. 81–92. URL: http://sites.computer.org/debull/A16mar/p81.pdf.

[19] V. Balegas, S. Duarte, C. Ferreira, N. Preguiça, and R. Rodrigues. "Making Weak Consistency Great Again". In: *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*. PaPoC '16. London, United Kingdom: ACM, 2016, 7:1–7:3. URL: http://doi.acm.org/10.1145/2911151.2911167.

[20] C. Baquero and F. Moura. "Using Structural Characteristics for Autonomous Operation". In: *SIGOPS Oper. Syst. Rev.* 33.4 (1999), pp. 90–96. URL: http://doi.acm.org/10.1145/334598.334614.

[21] D. Barbará-Millá and H. Garcia-Molina. "The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems". In: *The VLDB Journal* 3.3 (1994), pp. 325–353. URL: http://dx.doi.org/10.1007/BF01232643.

[22] Basho. *Riak*. Accessed Nov/2016. 2016. URL: http://basho.com/riak/.

[23] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. "A Critique of ANSI SQL Isolation Levels". In: *SIGMOD Rec.* 24.2 (1995), pp. 1–10. URL: http://doi.acm.org/10.1145/568271.223785.

[24] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[25] K. Birman and T. Joseph. "Exploiting Virtual Synchrony in Distributed Systems". In: *SIGOPS Oper. Syst. Rev.* 21.5 (1987), pp. 123–138. URL: http://doi.acm.org/10.1145/37499.37515.

[26] E. Brewer. "CAP Twelve Years Later: How the "Rules" Have Changed". In: *Computer* 45.2 (2012), pp. 23–29.

[27] E. A. Brewer. "Towards Robust Distributed Systems (Abstract)". In: *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*. PODC '00. Portland, Oregon, USA: ACM, 2000, pp. 7–. URL: http://doi.acm.org/10.1145/343477.343502.

[28] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. "From Session Causality to Causal Consistency." In: *Prooceding of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. IEEE Computer Society, 2004, pp. 152–158. URL: http://dblp.uni-trier.de/db/conf/pdp/pdp2004.html#BrzezinskiSW04.

[29] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Trans. Comput. Syst.* 26.2 (2008), 4:1–4:26. URL: http://doi.acm.org/10.1145/1365815.1365816.

[30] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. "VCC: A Practical System for Verifying Concurrent C". In: *Theorem Proving in Higher Order Logics*. Springer, 2009, pp. 23–42.

[31] *Companies Using NoSQL*. Accessed May-2016. URL: http://basho.com/about/customers/.

[32] *Companies Using NoSQL Apache Cassandra*. Accessed May-2016. URL: http://www.planetcassandra.org/companies/.

[33] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. "Logic and Lattices for Distributed Programming". In: *Proceedings of the 3rd ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: ACM, 2012, 1:1–1:14. URL: http://doi.acm.org/10.1145/2391229.2391230.

[34] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. "PNUTS: Yahoo!'s Hosted Data Serving Platform". In: *Proc. VLDB Endow.* 1.2 (2008), pp. 1277–1288. URL: http://dx.doi.org/10.14778/1454159.1454167.

[35] J. C. Corbett et al. "Spanner: Google's Globally-distributed Database". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI '12. Hollywood, CA, USA: USENIX Association, 2012, pp. 251–264. URL: http://dl.acm.org/citation.cfm?id=2387880.2387905.

[36] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, 2011.

[37] T. P. P. Council. *TPC Benchmark*. Accessed Nov/2016. 2016. URL: http://www.tpc.org/.

[38] S. B. Davidson, H. Garcia-Molina, and D. Skeen. "Consistency in a Partitioned Network: A Survey". In: *ACM Comput. Surv.* 17.3 (1985), pp. 341–370. URL: http://doi.acm.org/10.1145/5505.5508.

[39] L. De Moura and N. Bjørner. "Z3: An efficient SMT solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. TACAS '08. Springer, 2008, pp. 337–340.

[40] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. URL: http://doi.acm.org/10.1145/1294261.1294281.

[41] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. *Extended static checking*. Tech. rep. 159. Compaq Systems Research Center, 1998.

[42] P. Dixon. *Shopzilla's Site Redo - You Get What You Measure. Presented at Velocity Web Performance and Operations Conference*. 2009.

[43] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. "FaRM: Fast Remote Memory". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI '14. Seattle, WA: USENIX Association, 2014, pp. 401–414. URL: http://dl.acm.org/citation.cfm?id=2616448.2616486.

[44] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. "Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks". In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, 2013, 11:1–11:14. URL: http://doi.acm.org/10.1145/2523616.2523628.

[45] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. "GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks". In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. Seattle, WA, USA: ACM, 2014, 4:1–4:13. URL: http://doi.acm.org/10.1145/2670979.2670983.

[46] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. "The Daikon System for Dynamic Detection of Likely Invariants". In: *Sci. Comput. Program.* 69.1-3 (2007), pp. 35–45. URL: http://dx.doi.org/10.1016/j.scico.2007.01.015.

[47] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. "Making Snapshot Isolation Serializable". In: *ACM Trans. Database Syst.* 30.2 (2005), pp. 492–528. URL: http://doi.acm.org/10.1145/1071610.1071615.

[48] C. Flanagan and K. R. M. Leino. "Houdini, an Annotation Assistant for ESC/Java". In: *Proceeding of the International Symposium of Formal Methods for Increasing Software Productivity*. Ed. by J. N. Oliveira and P. Zave. FME '01. Berlin, Germany: Springer Berlin Heidelberg, 2001, pp. 500–517. URL: http://dx.doi.org/10.1007/3-540-45251-6_29.

[49] *Fusion Ticket*. Accessed September-2016. URL: http://www.fusionticket.org/.

[50] H. Garcia-Molina. "Using Semantic Knowledge for Transaction Processing in a Distributed Database". In: *ACM Trans. Database Syst.* 8.2 (1983), pp. 186–213. URL: http://doi.acm.org/10.1145/319983.319985.

[51] H. Garcia-Molina and K. Salem. "Sagas". In: SIGMOD '87 (1987), pp. 249–259. URL: http://doi.acm.org/10.1145/38713.38742.

[52] D. K. Gifford. "Weighted Voting for Replicated Data". In: *Proceedings of the 7th ACM Symposium on Operating Systems Principles*. SOSP '79. Pacific Grove, California, USA: ACM, 1979, pp. 150–162. URL: http://doi.acm.org/10.1145/800215.806583.

[53] S. Gilbert and N. Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services". In: *SIGACT News* 33.2 (2002), pp. 51–59. URL: http://doi.acm.org/10.1145/564585.564601.

[54] Gomez. *Why Web Performance Matters: Is Your Site Driving Customers Away?* Accessed Nov/2016. 2016. URL: http://www.mcrinc.com/Documents/Newsletters/201110_why_web_performance_matters.pdf.

[55] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. "'Cause I'M Strong Enough: Reasoning About Consistency Choices in Distributed Systems". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: ACM, 2016, pp. 371–384. URL: http://doi.acm.org/10.1145/2837614.2837625.

[56] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. "Readings in Database Systems". In: ed. by M. Stonebraker. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988. Chap. Granularity of Locks and Degrees of Consistency in a Shared Data Base, pp. 94–121. URL: http://dl.acm.org/citation.cfm?id=48751.48758.

[57] J. Gray and L. Lamport. "Consensus on Transaction Commit". In: *ACM Trans. Database Syst.* 31.1 (2006), pp. 133–160. URL: http://doi.acm.org/10.1145/1132863.1132867.

[58] J. Gray, P. Helland, P. O'Neil, and D. Shasha. "The Dangers of Replication and a Solution". In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. SIGMOD '96. Montreal, Quebec, Canada: ACM, 1996, pp. 173–182. URL: http://doi.acm.org/10.1145/233269.233330.

[59] P. G. D. Group. *Replication, Clustering, and Connection Pooling*. Accessed Nov/2016. 2016. URL: https://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling#Replication.

[60] T. Haerder and A. Reuter. "Principles of Transaction-oriented Database Recovery". In: *ACM Comput. Surv.* 15.4 (1983), pp. 287–317. URL: http://doi.acm.org/10.1145/289.291.

[61] P. Helland. "Life Beyond Distributed Transactions: an Apostate's Opinion". In: *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research*. CIDR '07. Asilomar, CA, USA, 2007, pp. 132–141. URL: http://www.cidrdb.org/cidr2007/papers/cidr07p15.pdf.

[62] P. Helland and D. Campbell. "Building on Quicksand". In: *Proocedings of the Fourth Biennial Conference on Innovative Data Systems Research*. CIDR '09. Asilomar, CA, USA, 2009.

[63] M. Herlihy and E. Koskinen. "Transactional Boosting: A Methodology for Highly-concurrent Transactional Objects". In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '08. Salt Lake City, UT, USA: ACM, 2008, pp. 207–216. URL: http://doi.acm.org/10.1145/1345206.1345237.

[64] M. P. Herlihy and J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 463–492. URL: http://doi.acm.org/10.1145/78969.78972.

[65] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[66] B. Holt, J. Bornholt, I. Zhang, D. Ports, M. Oskin, and L. Ceze. "Disciplined Inconsistency with Consistency Types". In: *Proceedings of the 7th ACM Symposium on Cloud Computing*. SoCC '16. Santa Clara, CA, USA: ACM, 2016, pp. 279–293. URL: http://doi.acm.org/10.1145/2987550.2987559.

[67] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. "An Analysis of Facebook Photo Caching". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 167–181. URL: http://doi.acm.org/10.1145/2517349.2522722.

[68] H. B. Hunt and D. J. Rosenkrantz. "The Complexity of Testing Predicate Locks". In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. SIGMOD '79. Boston, Massachusetts: ACM, 1979, pp. 127–133. URL: http://doi.acm.org/10.1145/582095.582115.

[69] A. Inc. *Amazon Elastic Compute Cloud (EC2)*. Accessed Jan/2017. 2017. URL: https://aws.amazon.com/ec2/.

[70] G. Inc. *Google Cloud Platform: APP Engine*. Accessed Jan/2017. 2017. URL: https://cloud.google.com/appengine/.

[71] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. "VeriFast: a Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *NASA Formal Methods*. Springer, 2011, pp. 41–55.

[72] J. Jing, A. S. Helal, and A. Elmagarmid. "Client-server Computing in Mobile Environments". In: *ACM Comput. Surv.* 31.2 (1999), pp. 117–157. URL: http://doi.acm.org/10.1145/319806.319814.

[73] J. J. Kistler and M. Satyanarayanan. "Disconnected Operation in the Coda File System". In: *ACM Trans. Comput. Syst.* 10.1 (1992), pp. 3–25. URL: http://doi.acm.org/10.1145/146941.146942.

[74] R. Klophaus. "Riak Core: Building Distributed Applications Without Shared State". In: *Proocedings of the ACM SIGPLAN Commercial Users of Functional Programming*. CUFP '10. Baltimore, Maryland: ACM, 2010, 14:1–14:1. URL: http://doi.acm.org/10.1145/1900160.1900176.

[75] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. "Consistency Rationing in the Cloud: Pay Only when It Matters". In: *Proc. VLDB Endow.* 2.1 (2009), pp. 253–264. URL: http://dl.acm.org/citation.cfm?id=1687627.1687657.

[76] H. T. Kung and J. T. Robinson. "On Optimistic Methods for Concurrency Control". In: *ACM Trans. Database Syst.* 6.2 (1981), pp. 213–226. URL: http://doi.acm.org/10.1145/319566.319567.

[77] R. Ladin, B. Liskov, and L. Shrira. "Lazy Replication: Exploiting the Semantics of Distributed Services". In: *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*. PODC '90. Quebec City, Quebec, Canada: ACM, 1990, pp. 43–57. URL: http://doi.acm.org/10.1145/93385.93399.

[78] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (2010), pp. 35–40. URL: http://doi.acm.org/10.1145/1773912.1773922.

[79] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), pp. 558–565. URL: http://doi.acm.org/10.1145/359545.359563.

[80] L. Lamport. "The Temporal Logic of Actions". In: *ACM Trans. Program. Lang. Syst.* 16.3 (1994), pp. 872–923. URL: http://doi.acm.org/10.1145/177492.177726.

[81] L. Lamport. "The Part-time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (1998), pp. 133–169. URL: http://doi.acm.org/10.1145/279227.279229.

[82] L. Lamport. "Paxos Made Simple". In: *SIGACT News* 32.4 (2001), pp. 51–58. URL: http://research.microsoft.com/users/lamport/pubs/paxos-simple.pdf.

[83] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. "Making Geo-replicated Systems Fast As Possible, Consistent when Necessary". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI '12. Hollywood, CA, USA: USENIX Association, 2012, pp. 265–278. URL: http://dl.acm.org/citation.cfm?id=2387880.2387906.

[84] C. Li, J. a. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. "Automating the Choice of Consistency Levels in Replicated Systems". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. ATC '14. Philadelphia, PA: USENIX Association, 2014, pp. 281–292. URL: http://dl.acm.org/citation.cfm?id=2643634.2643664.

[85] A. van der Linde, J. a. Leitão, and N. Preguiça. "Delta-CRDTs: Making Delta-CRDTs Delta-based". In: *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*. PaPoC '16. London, United Kingdom: ACM, 2016, 12:1–12:4. URL: http://doi.acm.org/10.1145/2911151.2911163.

[86] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "Don'T Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS". In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 401–416. URL: http://doi.acm.org/10.1145/2043556.2043593.

[87] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "Stronger Semantics for Low-latency Geo-replicated Storage". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. NSDI '13. Lombard, IL: USENIX Association, 2013, pp. 313–328. URL: http://dl.acm.org/citation.cfm?id=2482626.2482657.

[88] P. Mahajan, L. Alvisi, and M. Dahlin. *Consistency, Availability, Convergence*. Tech. rep. TR-11-22. Computer Science Department, University of Texas at Austin, 2011.

[89] S. Martin, M. Ahmed-Nacer, and P. Urso. "Controlled Conflict Resolution for Replicated Document". In: *Proocedings of the 8th International Conference on Collaborative Computing: Networking, Applications and Worksharing*. CollaborateCom '12. 2012, pp. 471–480.

[90] S. Martin, M. Ahmed-Nacer, and P. Urso. "Abstract Unordered and Ordered Trees CRDT". In: *CoRR* abs/1201.1784 (2012).

[91] M. Mayer. *In Search of... A better, faster, stronger Web. Presented at Velocity Web Performance and Operations Conference*. 2009.

[92] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd. "I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades". In: *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*. NSDI '17. Boston, MA: USENIX Association, 2017, pp. 453–468. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/mehdi.

[93] P. E. O'Neil. "The Escrow Transactional Method". In: *ACM Trans. Database Syst.* 11.4 (1986), pp. 405–430. URL: http://doi.acm.org/10.1145/7239.7265.

[94] D. Ongaro and J. Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. ATC '14. Philadelphia, PA: USENIX Association, 2014, pp. 305–320. URL: http://dl.acm.org/citation.cfm?id=2643634.2643666.

[95] Oracle. *Oracle Isolation Levels*. Accessed Nov/2016. 2016. URL: http://docs.oracle.com/cd/B12037_01/server.101/b10743/consist.htm.

[96] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. "The RAMCloud Storage System". In: *ACM Trans. Comput. Syst.* 33.3 (2015), 7:1–7:55. URL: http://doi.acm.org/10.1145/2806887.

[97] S. S. Owicki. "Axiomatic Proof Techniques for Parallel Programs." AAI7612884. PhD thesis. Ithaca, NY, USA, 1975.

[98] G. Pang, T. Kraska, M. J. Franklin, and A. Fekete. "PLANET: Making Progress with Commit Processing in Unpredictable Environments". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: ACM, 2014, pp. 3–14. URL: http://doi.acm.org/10.1145/2588555.2588558.

[99] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. "Detection of Mutual Inconsistency in Distributed Systems". In: *IEEE Trans. Softw. Eng.* 9.3 (1983), pp. 240–247. URL: http://dx.doi.org/10.1109/TSE.1983.236733.

[100] D. L. Parnas. "Precise Documentation: The Key to Better Software". In: *The Future of Software Engineering*. Ed. by S. Nanz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 125–148. URL: http://dx.doi.org/10.1007/978-3-642-15187-3_8.

[101] C. Plattner and G. Alonso. "Ganymed: Scalable Replication for Transactional Web Applications". In: *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*. Ed. by H.-A. Jacobsen. Middleware '04. Toronto, Canada: Springer Berlin Heidelberg, 2004, pp. 155–174. URL: http://dx.doi.org/10.1007/978-3-540-30229-2_9.

[102] D. R. K. Ports and K. Grittner. "Serializable Snapshot Isolation in PostgreSQL". In: *Proc. VLDB Endow.* 5.12 (2012), pp. 1850–1861. URL: http://dx.doi.org/10.14778/2367502.2367523.

[103] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia. "A Commutative Replicated Data Type for Cooperative Editing". In: *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*. ICDCS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 395–403. URL: http://dx.doi.org/10.1109/ICDCS.2009.20.

[104] N. M. Preguiça, C. Baquero, F. Moura, J. L. Martins, R. Oliveira, H. J. a. L. Domingos, J. O. Pereira, and S. Duarte. "Mobile Transaction Management in Mobisnap". In: *Proceedings of the East-European Conference on Advances in Databases and Information Systems Held Jointly with International Conference on Database Systems*

for Advanced Applications: Current Issues in Databases and Information Systems. ADBIS-DASFAA '00. London, UK: Springer-Verlag, 2000, pp. 379–386. URL: http://dl.acm.org/citation.cfm?id=646044.676434.

[105] D. Pritchett. "BASE: An Acid Alternative". In: *Queue* 6.3 (2008), pp. 48–55. URL: http://doi.acm.org/10.1145/1394127.1394128.

[106] K. Ramamritham and C. Pu. "A Formal Characterization of Epsilon Serializability". In: *IEEE Trans. on Knowl. and Data Eng.* 7.6 (1995), pp. 997–1007. URL: http://dx.doi.org/10.1109/69.476504.

[107] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. "Resolving File Conflicts in the Ficus File System". In: *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*. USTC '94. Boston, Massachusetts: USENIX Association, 1994, pp. 12–12. URL: http://dl.acm.org/citation.cfm?id=1267257.1267269.

[108] R. van Renesse and F. B. Schneider. "Chain Replication for Supporting High Throughput and Availability". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI '04. San Francisco, CA: USENIX Association, 2004, pp. 7–7. URL: http://dl.acm.org/citation.cfm?id=1251254.1251261.

[109] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. "Replicated Abstract Data Types: Building Blocks for Collaborative Applications". In: *J. Parallel Distrib. Comput.* 71.3 (2011), pp. 354–368. URL: http://dx.doi.org/10.1016/j.jpdc.2010.12.006.

[110] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. "The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1311–1326. URL: http://doi.acm.org/10.1145/2723372.2723720.

[111] E. Schurman and J. Brutlag. *Performance Related Changes and their User Impact. Presented at Velocity Web Performance and Operations Conference*. 2009.

[112] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *A Comprehensive Study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA, 2011, p. 50. URL: https://hal.inria.fr/inria-00555588.

[113] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-free Replicated Data Types". In: *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Ed. by X. Défago, F. Petit, and V. Villain. Vol. 6976. SSS '11. Grenoble, France: Springer, 2011, pp. 386–400.

[114] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. "Transaction Chopping: Algorithms and Performance Studies". In: *ACM Trans. Database Syst.* 20.3 (1995), pp. 325–363. URL: http://doi.acm.org/10.1145/211414.211427.

[115] L. Shrira, H. Tian, and D. Terry. "Exo-leasing: Escrow Synchronization for Mobile Clients of Commodity Storage Servers". In: *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Middleware '08. Leuven, Belgium: Springer-Verlag New York, Inc., 2008, pp. 42–61. URL: http://dl.acm.org/citation.cfm?id=1496950.1496954.

[116] S. Sivasubramanian. "Amazon dynamoDB: A Seamlessly Scalable Non-relational Database Service". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: ACM, 2012, pp. 729–730. URL: http://doi.acm.org/10.1145/2213836.2213945.

[117] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. "Transactional Storage for Geo-replicated Systems". In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 385–400. URL: http://doi.acm.org/10.1145/2043556.2043592.

[118] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications". In: *SIGCOMM Comput. Commun. Rev.* 31.4 (2001), pp. 149–160. URL: http://doi.acm.org/10.1145/964723.383071.

[119] M. Stonebraker. "Concurrency Control and Consistency of Multiple Copies of Data in Distributed Ingres". In: *IEEE Trans. Softw. Eng.* 5.3 (1979), pp. 188–194. URL: http://dx.doi.org/10.1109/TSE.1979.234180.

[120] C. Sun and C. Ellis. "Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements". In: *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*. CSCW '98. Seattle, Washington, USA: ACM, 1998, pp. 59–68. URL: http://doi.acm.org/10.1145/289444.289469.

[121] B. Technologies. *Data Types*. Accessed Jan/2017. 2017. URL: http://docs.basho.com/riak/kv/2.2.0/learn/concepts/crdts/.

[122] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System". In: *SIGOPS Oper. Syst. Rev.* 29.5 (1995), pp. 172–182. URL: http://doi.acm.org/10.1145/224057.224070.

[123] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. "Session Guarantees for Weakly Consistent Replicated Data". In: *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*. PDIS '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 140–149. URL: http://dl.acm.org/citation.cfm?id=645792.668302.

[124] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. "Consistency-based Service Level Agreements for Cloud Storage". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 309–324. URL: http://doi.acm.org/10.1145/2517349.2522731.

[125] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. "Speedy Transactions in Multicore In-memory Databases". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 18–32. URL: http://doi.acm.org/10.1145/2517349.2522713.

[126] W. Vogels. "Eventually Consistent". In: *Commun. ACM* 52.1 (2009), pp. 40–44. URL: http://doi.acm.org/10.1145/1435417.1435432.

[127] G. D. Walborn and P. K. Chrysanthis. "Supporting Semantics-based Transaction Processing in Mobile Database Applications". In: *Proceedings of the 14th Symposium on Reliable Distributed Systems*. SRDS '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 31–. URL: http://dl.acm.org/citation.cfm?id=829520.830874.

[128] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. "Salt: Combining ACID and BASE in a Distributed Database". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI '14. Broomfield, CO: USENIX Association, 2014, pp. 495–509. URL: http://dl.acm.org/citation.cfm?id=2685048.2685087.

[129] H. Yu and A. Vahdat. "Design and Evaluation of a Continuous Consistency Model for Replicated Services". In: *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*. OSDI '00. San Diego, California: USENIX Association, 2000, pp. 21–21. URL: http://dl.acm.org/citation.cfm?id=1251229.1251250.

[130] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. "Write Fast, Read in the Past: Causal Consistency for Client-Side Applications". In: *Proceedings of the 16th Annual Middleware Conference*. Middleware '15. Vancouver, BC, Canada: ACM, 2015, pp. 75–87. URL: http://doi.acm.org/10.1145/2814576.2814733.

[131] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. "Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 276–291. URL: http://doi.acm.org/10.1145/2517349.2522729.